P1 Fundamentals

P1 Fundamentals

Personal Python Method Reference
https://docs.google.com/document/
d/1PlwubfOE6Q443dIG651ZbH1Obyx6eJ3jETKrGS166j8/edit

All W3 Schools Sections
https://www.w3schools.com/python/python_strings_methods.asp

Official Python docs
https://docs.python.org/3.12/library/index.html

Shift Tab will Unindent highlighted lines in VS Code

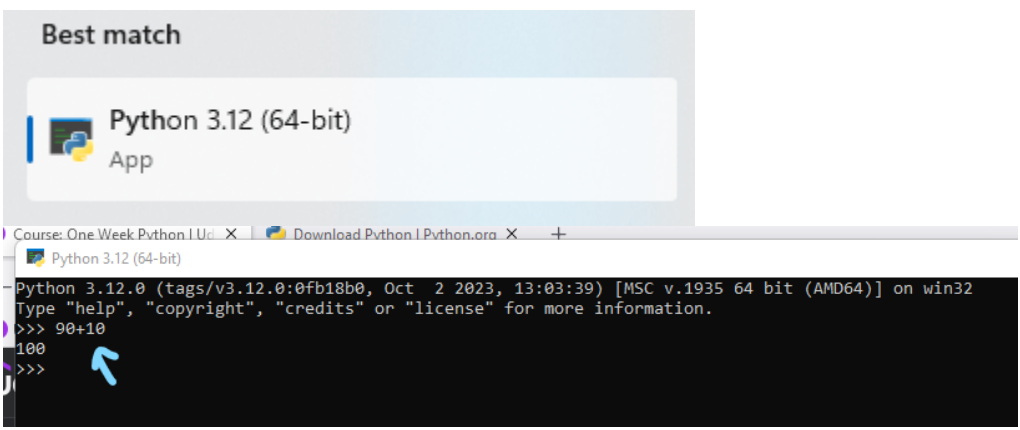
Environments & Setup

REPL Environment

Regular REPL Env

Just like Node.js, Python has a REPL Environment where you can run code quickly
Real applications are not built this way

1. Search python in your Windows searchbar then select it
2. Type in Python one line at a time and get outputs right away


Best match
Python 3.12 (64-bit)
App
Python 3.12 (64-bit)
Python 3.12.0 (tags/v3.12.0:0fb18b0, Oct 2 2023, 13:03:39) [MSC v.1935 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 90+10
100
>>>

iPython REPL

Assuming iPython is installed

1. Open Windows terminal
2. Enter `ipython` then treat it like a regular REPL
   (Has bonus features like autocomplete and color coating)

Playgrounds

You can write 1 Python script and run it
Not limited to writing tiny scripts like you are with the REPL

https://www.online-python.com/KXmLvup8J6

Dependencies

1. Install Python language (make sure to get a recent version since your computer might already come with 2.0 preinstalled)
   https://www.python.org/downloads/

2. Install Python extension in VS Code - make sure interpreter is set to Python 3 (current mainstream version)

3. Install iPython in your windows terminal

```
pip install ipython
```

Definitions

ITERABLE:
any object capable of returning its members one at a time, allowing it to be iterated over in a loop

1. Lists
   Ordered, mutable collections of items.

2. Tuples
   Ordered, immutable collections of items.

3. Strings
   Sequence of characters.

4. Dictionaries
   Collections of key-value pairs.

5. Sets
   Unordered collections of unique items.

6. Frozensets
   Immutable and hashable versions of sets.

7. Range objects
   Produced by the `range()` function, often used for looping a specific number of times in `for` loops.

8. Bytes and Bytearrays
   Sequences of bytes, where `bytes` is immutable and `bytearray` is mutable.

9. Generators
   Functions that yield a sequence of values using the `yield` statement.

10. Files
    File objects returned by the `open()` function can be iterated over to read lines one by one.

11. Additionally, many modules and libraries define their own iterable objects

    - For instance, in NumPy, arrays are iterables
    - In Pandas, DataFrame and Series objects are iterable

Big O Scores for Python Methods

Lists

How to Remember the Scores

TIME COMPLEXITY

- If a method needs to run through the entire list or causes reindexing, the time complexity is likely `O(n)`
  With the exception of sort, which is `O(n log n)` due to using a special sorting algorithm under the hood
- Otherwise the time complexity is `O(1)`

SPACE COMPLEXITY

- If the method creates a new list, the time complexity is `O(n)`
- Otherwise the space complexity is `O(1)`

Main Table

When indexing is required, the score is O(n) - and when it isn't the score is O(1)

Dictionaries

In practice, many big O scores are O(1)
If n is known and fixed, O(n) becomes O(1) in practice
Ex You use a dict or counter to make a frequency object with 52 kvps for a-z + A-Z chars

How to Remember the Scores

TIME COMPLEXITY

- O(n) if a method creates a new dict, unpacks an existing one, merges 2, or compares 2
- O(1) otherwise

SPACE COMPLEXITY (same as above)

- O(n) if a method creates a new dict, unpacks an existing one, or compares 2
- O(1) otherwise
- Only 1 difference compared to space complexity
  Merging dictionaries with update() has O(1) space complexity since you reuse an already existing dict

Standard Table

O(1)* means:

- Usually constant time (O(1)) in practice
- BUT could theoretically degrade to O(n) in worst case due to hash collisions

Iterator Use Cases

Try to use iterators when you can to save some performance

Counter Dict Methods

Strings

Table

Note: Strings in Python are immutable sequences of characters

Where:

- a is length of first string (s1)
- b is length of second string (s2)
- n is length of string or multiplier

Key characteristics:

1. Immutable like tuples
2. All methods that appear to modify the string actually return new strings
3. String slicing creates new strings
4. Common operations like concatenation create new strings

Sets

Sets use hash tables, so operations marked with * have O(1) average case but O(n) worst case due to potential hash collisions.

Where:

- a is size of first set (s1)
- b is size of second set (s2)
- Average case is O(1), worst case is O(n) due to potential hash collisions

Tuples >

Where:

- a is size of first tuple (t1)
- b is size of second tuple (t2) or multiplier (n)

What are Iterators?

Usually it's more efficient to use Python methods that return an iterator instead of a more common data structure

EXAMPLE:
Use an iterator to get a reversed set of values from my_list

```
my_list = [1, 3, 6, 9]
reversed_list_iterator = reversed(my_list) # O(1) WHICH IS
for num in reversed_list_iterator:
    print(num)

reversed_list = my_list[::-1] # O(n) WHICH IS WORSE
for num in reversed_list:
    print(num)
```

```
9
6
3
1
```

```
9
6
3
1
```

ss

= = = = = = = = = BASICS

= = = = = = = = =

Python Fundamentals

Variables

How they Work

Variables allow you to store certain variables in memory and reference them later

- You can change their values using assignment operators like `=`, `+=`, `*=` ... etc
- Using variables in expressions will not change their values

```
x = 6
print(x) # 6
x * 10 # expression that equals 60 but does not redefine th
print(x) # 6
x = x + 2
print(x) # 8
```

Naming Errors

RULES WHEN NAMING VARIABLES

- No numbers at the very beginning
- No spaces - use underscores instead
- No using reserved keywords as names (ex. False, def)

```
Here is a list of the Python keywords.  Enter any keyword to get more help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not
```

Variable Naming Conventions

Snake Case is the Standard

Python developers generally use snake_case to name their variables that include multiple words
Camel case can be used, but is not as common as it is in JS

```
player_one = "Jimothy Cool" # preferred
playerOne = "Jimothy Cool"  # allowed but not encouraged
```

Capital Names for Const Variables

Python has no variables whose values are unchangeable

- Instead, Python devs will capitalize a variable name to signal that this variable should not have it's value changed
- They still can be changed, but as a dev you should recognize that changing a variable with all caps will cause problems

```
STANDARD = "misc"
STANDARD = "one" # really should be doing this since the va
print(STANDARD) # one
```

Reserved Keywords

You should not name your variables the same as reserved keywords - even if Python allows you to without throwing an error

Doing so will cut off certain functionalities

List of Reserved Keywords

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Why we Avoid Reserved Keywords when Naming Variables

EXAMPLE:
Naming a variable str will prevent you from type casting to strings

Ok variable name

```
str1 = "fun"
print(str(20)) # "20"
```

Bad variable name

```
str = "fun" # str is a casting method, and we are overridin
print(str(20)) # TypeError: 'str' object is not callable
```

print() Method

The `print()` method logs messages to the terminal and returns nothing- which is useful for debugging

- Place any variable or expression inside the brackets, and their value will be logged
- You can place multiple variables or expressions in the brackets if you

wish

```
x = "6ix"
print(1, 2, 3, x)
```

pass keyword: Placeholder Code

The pass keyword causes nothing to happen

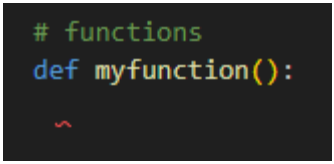
- It exists to stop Python from throwing errors when you have nothing in a location where code is supposed to be
- Use it to satisfy syntax checks, then replace it with real code later on

EXAMPLES:

```
# functions
def myfunction():
  pass
# loops
for x in [0, 1, 2]:
  pass
# conditionals
a = 33
b = 200
if b > a:
  pass
```

```
# classes
class Person:
  pass
```

Without pass, we'd get syntax errors like this one (notice the red squiggle)



input() : Collect User Inputs from Terminal

Purpose

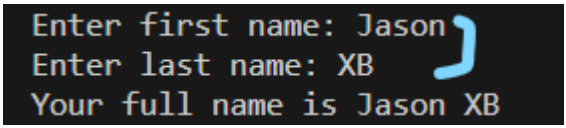
1. Asks the user a question via the command line
2. Lets the user type in an answer and hit enter
3. You can save the answer to a variable and use it throughout the rest of your script

ALL USER INPUTS WILL BECOME STRINGS
Even if the user types `1` with no quotations, the input method will accept it as a string - just like input fields in HTML

Demo

Try running this script to see how the inputs work

```
greeting = "Welcome to the name app!"
q1 = input("Enter first name: ") # must enter a value befor
q2 = input("Enter last name: ") # must enter a value before
print(f"Your full name is {q1} {q2}")
```



Practical Example

Remember how `npm init` works when creating projects
Asks for project name, version number, then generates a package.json using some of the answers you gave

Method Chaining

You can reduce the number of variables used if you chain methods to the end of each other
Just make sure that a method returns data of the tpe that the next method in the chain requires

Example: Chain 2 string methods together

Long

```
email = "mason  XD@gmail.com"
no_space_email = email.replace(" ", "") # replace returns a
lowercase_email = no_space_email.lower() # lower accepts a
print(lowercase_email) # masonxd@gmail.com
```

Shortened using chaining

```
email = "mason  XD@gmail.com"
formatted_email = email.replace(" ", "").lower() # 2 chaine
print(formatted_email) # masonxd@gmail.com
```

Shallow + Deep Cloning

Shallow Clone

Changes to nested data structures are synchronized - meaning changes to the original's nested properties will affect the clone, and vice versa

```
person1 = {
    "name": "Ben Shaw",
    "age": 16,
    "profile_data": {"employee_num": 10004, "position": "As
}
person2 = person1.copy() # shallow clone
print("SHALLOW CLONE SHOWCASE")
person1["name"] = "Alex" # change a nested property
print(person1)
print(person2)
print()
print("PROOF THAT THIS IS NOT A DEEP CLONE")
person1["profile_data"]["position"] = "Burger Flipper"
print(person1)
print(person2) # position would not be "Burger Flipper" if
```

```
SHALLOW CLONE SHOWCASE
{'name': 'Alex', 'age': 16, 'profile_data': {'employee_num': 10004, 'position': 'Assistant Manager'}}
{'name': 'Ben Shaw', 'age': 16, 'profile_data': {'employee_num': 10004, 'position': 'Assistant Manager'}}

PROOF THAT THIS IS NOT A DEEP CLONE
{'name': 'Alex', 'age': 16, 'profile_data': {'employee_num': 10004, 'position': 'Burger Flipper'}}
{'name': 'Ben Shaw', 'age': 16, 'profile_data': {'employee_num': 10004, 'position': 'Burger Flipper'}}
```

Deep Clone

Requires an import

```
import copy
person1 = {
    "name": "Ben Shaw",
    "age": 16,
    "profile_data": {"employee_num": 10004, "position": "As
}
person2 = copy.deepcopy(person1) # shallow clone
print("SHALLOW CLONE SHOWCASE")
person1["name"] = "Alex" # change a nested property
print(person1)
print(person2)
print("PROOF THAT THIS IS NOT A DEEP CLONE")
person1["profile_data"]["position"] = "Burger Flipper"
print(person1)
print(person2)
```

```
SHALLOW CLONE SHOWCASE
{'name': 'Alex', 'age': 16, 'profile_data': {'employee_num': 10004, 'position': 'Assistant Manager'}}
{'name': 'Ben Shaw', 'age': 16, 'profile_data': {'employee_num': 10004, 'position': 'Assistant Manager'}}

PROOF THAT THIS IS NOT A DEEP CLONE
{'name': 'Alex', 'age': 16, 'profile_data': {'employee_num': 10004, 'position': 'Burger Flipper'}}
{'name': 'Ben Shaw', 'age': 16, 'profile_data': {'employee_num': 10004, 'position': 'Assistant Manager'}}
```

Safely Accessing Values from all Data Structures

Trying to access data no present inside a data structure can sometimes produce errors
There are ways to negate this

Access list item that's out of bounds (causes error)

```
my_list = [0, 1]
```

```
a = my_list[5]  #! IndexError: list index out of range
a = my_list[5] if 5 < len(my_list) else None  # Safe
```

Access dictionary KVP that does not exist using bracket notation (causes error)

```
my_dict = {"a": 1, "b": 2, "c": 3, "d": 4}
b = my_dict["z"]  #! KeyError: 'z'
b = my_dict.get("z", "not found")  # "not found" (safe)
```

Access tuple item that is out of bounds (causes error)

```
my_tuple = ("a", "b")
c = my_tuple[5] #! IndexError: tuple index out of range
c = my_tuple[5] if 5 < len(my_tuple) else None  # None
```

Python Methods are Upper Bound Exclusive

Definition

If a Python method has parameters that represent indexes, it's going to be upper bound exclusive

- Means that if you supply a start and end index, the end index will be excluded from the operation
- In human terms "do X up to but not including the end index"

```
my_list = [0, 1, 2, 3, 4]
# slice from index 1 up to (but not including) index 4
print(my_list[1:4])  # Output: [1, 2, 3]
```

One Exception: range() with a Negative Step

Technically this isn't a method with a range index, but it does affect the number of iterations

Forward Step (upper bound exclusive)

```
for i in range(4):
    print(i)  # Outputs: 0 1 2 3

# Clearly upper bound exclusive since it iterates up to but
```

Backwards Step

- Upper bound becomes inclusive (why we see it log 4)
- Lower bound becomes exclusive (why 0 is not logged anymore)

```
# Count down from 5 to 1
for i in range(4, 0, -1):
    print(i)  # Outputs: 4, 3, 2, 1
```

In a way, Python remains upper bound exclusive 100% of the time
It just flips the position of the upper bound when doing a range backwards

Functions

Functions in Python

Basics

STANDARD SYNTAX:

```
def divideNums(x, y):
    return x/y # indentation required
reusable_variable = divideNums(12, 4)
print(reusable_variable) # 3
```

DEFINE A FUNCTION BEFORE YOU CALL IT:
There are some tricky workarounds to avoid this limitation, but that's a more advanced topic we may or may not cover later

```
print(divideNums(12)) # CAUSES AN ERROR
def divideNums(x, y=1):
    return x/y
```

```
PS C:\Users\jason\OneDrive\Desktop\Playgrounds\python_basic> &
Python312/python.exe c:/Users/jason/OneDrive/Desktop/Playgrounc
Traceback (most recent call last):
  File "c:\Users\jason\OneDrive\Desktop\Playgrounds\python_basi
    print(divideNums(12)) # 12
          ^^^^^^^^^^
NameError: name 'divideNums' is not defined
```

Regular Parameters are Mandatory

- If you define a function parameter, calling the function without passing in a value for this parameter will cause a fatal error
  Exception - default parameters

- This is not like JS where the function will accept the lack of a parameter as `undefined`

```python
def divideNums(x, y):
    if not x:
        return "x argument not provided" # not even reached
    if not y:
        return "y argument not provided" # not even reached
    return x/y
print(divideNums(12))
```

```
Traceback (most recent call last):
  File "c:\Users\jason\OneDrive\Desktop\Playgrounds\python_basic\first.py", line 9, in <module>
    print(divideNums(12))
          ^^^^^^^^^^^^^
TypeError: divideNums() missing 1 required positional argument: 'y'
PS C:\Users\jason\OneDrive\Desktop\Playgrounds\python basic>
```

Outputs None if no return is Reached

If your function executes and a `return` keyword is not reached, the function returns a `None` value
You can also use `return None` deliberately if you wish

```python
def divideNums(x, y):
    if (y != 0):
        return x/y
print(divideNums(12,0))   # None
```

Skimmed Topics

You're already a seasoned JS developer, so certain things are obvious to you now

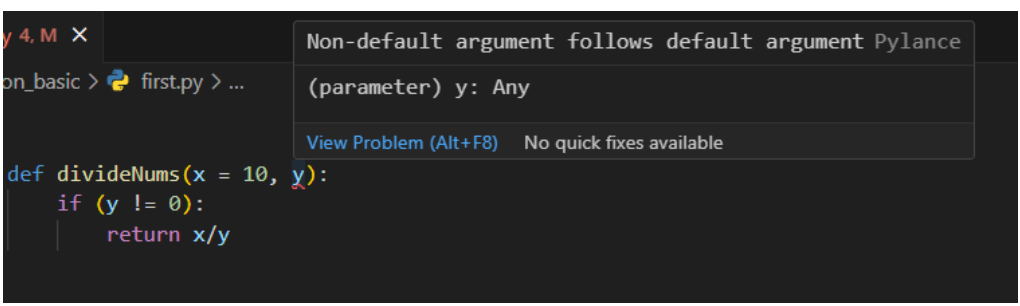
- Functions can have 0, 1, or multiple arguments
- The return keyword is used to end the execution of a function and to have it spit out a value

Default Parameters

You can define default values for Python functions - which makes some parameters optional

```
def divideNums(x, y=1):
    return x/y
print(divideNums(12)) # 12
```

Must define all your default parameters after the regular ones
If you don't, an error will be thrown



*args: Unspecified # of Arguments

Remaining Parameter Prefix: *

If you want to create a function that accepts an unspecified number of arguments, prefix the final argument with `*`

- All the remaining arguments (if any) fed when you call the function will be placed in a `tuple`
  You can access tuple items with index bracket syntax or iterate through them with a for loop

- You can have regular parameters defined before your `*param_name`
- Using `*args` is the typical convention, but you can name the parameter anything as long as it's prefixed by *
  Ex. `*remaining`, `\*leftovers`

Demos

Example 1: All arguments unspecified

```
def sum_up(*args):
    total = 0
    for num in args:
        total += num
    return total
result = sum_up(1, 2, 3) # 6
result = sum_up() # 0 (YOU ARE ALLOWED TO PASS *args 0 ARGU
```

Example 2: 1 regular argument, while the rest are unspecified

```
def multip(num1, *remaining):
    sum_of_remaining = 0
    for num in remaining:
        sum_of_remaining += num
    return num1 * sum_of_remaining
result = multip(10, 2, 3)  # 50 because 10 * (2+3)
```

Keyword Arguments

The way we regularly pass values to functions is referred to as `positional arguments`

Python provides an alternative way of passing arguments which are called `keyword arguments` or `named arguments`

How they Work

Keyword arguments let you namedrop which argument you're passing while you call a function

- You can namedrop arguments in any order you want
- Once you use a keyword argument, you are no longer allowed to use positional arguments while calling a function
- Can give keyword arguments a default value if you wish (see example in next subsection)

```
def priceCalculator(price, qty, tax):
    return (price * (1 + tax)) * qty
a = priceCalculator(price=50, qty=2, tax=0.05) # 105
b = priceCalculator(tax=0.05, price=50, qty=2) # 105
```

Mixed use of Keyword + Positional Arguments

You can use positional arguments THEN keyword arguments afterwards
You can NOT use keyword arguments after any positional arguments however

```
positional args → keyword args   ALLOWED

keyword args → positional args   SYNTAX ERROR
```

ACCEPTABLE:

```
def priceCalculator(price, qty=1, tax=0.13):
    return (price * (1 + tax)) * qty
# mix. positional into keyword arg
b = priceCalculator(1, tax=0.13)  # (output is 1.13)
```

NOT ACCEPTABLE

```
def priceCalculator(price, qty=1, tax=0.13):
    return (price * (1 + tax)) * qty
b = priceCalculator(tax=0.13, 1)  # SyntaxError: positional
```

**kwargs: Unspecified # of Keyword Arguments

Remaining keyword Parameter Prefix: **

Having `**` in front of a parameter in a function tells Python that it should create a dictionary containing all remaining keyword arguments as KVPs

- You are allowed to have regular arguments before `**kwargs`, but not after
- It's typical convention to use `**kwargs`, but any name works as long as it's prefixed by `**`
  Ex. `**remaining`

Demos

Example 1: All unspecified number of keyword args

```
def print_ages(**kwargs):
    for key, val in list(kwargs.items()): # list of tuples
        print(f"{key} is {val} years old")
print_ages(Marcy=39, Al=55, Bud=17)
# LOGS
# Marcy is 39 years old
# Al is 55 years old
# Bud is 17 years old
```

Example 2: Positional args first, then unspecified number of kewyrod args

```
def print_ages(show, **kwargs):
    for key, val in list(kwargs.items()):  # list of tuples
        print(f"{show}: {key} is {val} years old")
print_ages("Married with children", Marcy=39, Al=55, Bud=17
# Logs
# Married with children: Marcy is 39 years old
# Married with children: Al is 55 years old
# Married with children: Bud is 17 years old
```

Gacha: Mutable Default Params in Functions are Shared

When we define functions, we may specify a mutable data structure as a default value (list, dictionary... etc)

- This default parameter value is shared for every function call
  If we call a function a second time after the default param is mutated, it will not retain the starting value we specified
- This is only a problem if the default parameter is set to a mutable data structure
  Will not affect immutable strings, tuples, numbers... etc

Problem Demonstrated

You would expect the 2 logs to be the following - but they aren't thanks to this gacha

1. `{'Ben': 'Assistant Manager'}`
2. `{'Ben': 'Assistant Manager', 'Alex': 'Burger Flipper'}`

```
def addKVP(key, val, dict={}):
    dict[key] = val
    return dict
d1 = addKVP("Ben", "Assistant Manager")
print(d1) # {'Ben': 'Assistant Manager'}
# At this point, the dict param default value is not {}
# It is {'Ben': 'Assistant Manager'} since it has been muta
d2 = addKVP("Alex", "Burger Flipper")
print(d2) # {'Ben': 'Assistant Manager', 'Alex': 'Burger Fl
```

```
{'Ben': 'Assistant Manager'}
{'Ben': 'Assistant Manager', 'Alex': 'Burger Flipper'}
```

Workaround

Deep clone the default parameter at the start of the function before it's mutated by a method call

```
import copy
def addKVP(key, val, dict={}):
    dict = copy.deepcopy(dict)
    dict[key] = val
    return dict
d1 = addKVP("Ben", "Assistant Manager")
print(d1) # {'Ben': 'Assistant Manager'}
d2 = addKVP("Alex", "Burger Flipper")
```

```
print(d2) # {'Alex': 'Burger Flipper'}
```

```
{'Ben': 'Assistant Manager'}
{'Alex': 'Burger Flipper'}
```

Not a problem for Immutable Default Parameter Values

This problem won't occur when working with immutable default parameter values

```
def add_to_string(ext, txt="Employee: "):
    txt += ext
    return txt
d1 = add_to_string("Ben")
print(d1) # Employee: Ben
d2 = add_to_string("Alex")
print(d2) # Employee: Alex
```

```
Employee: Ben
Employee: Alex
```

Passing Values in a Sequential Data Structures

We can pass a sequential data structure to a method

List, Set, Tuple

```
nums = [1, 2, 3, 4, 5] # works with lists (output is 15)
nums = (1, 2, 3, 4, 5) # works with tuples (output is 15)
nums = {1, 2, 3, 4, 5} # works with sets (output is 15)
def add_all(*args):
    total = 0
    for num in args:
        total += num
    return total
d1 = add_all(*nums)
print(d1) # 15
```

Dictionary

Pass keys of a dictionary

```
pkmn = {1: "Bulbasaur", 2: "Squirtle", 3: "Charmander"}
def add_all(*args):
    print(args) # (1, 2, 3)
d1 = add_all(*pkmn)
```

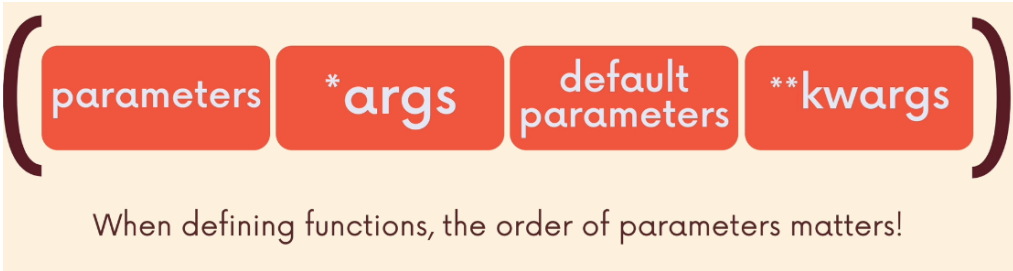
Pass values of a dictionary

```
pkmn = {1: "Bulbasaur", 2: "Squirtle", 3: "Charmander"}
def add_all(*args):
    print(args) # ('Bulbasaur', 'Squirtle', 'Charmander')
d1 = add_all(*pkmn.values())
```

Parameter List Ordering

This only comes into play if you're combining positional args, keyword args, default values, and having an unspecified amount of them

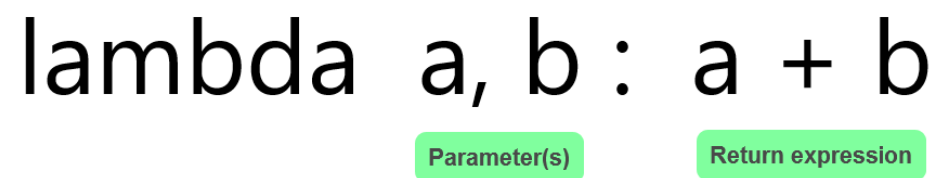
I've never had to do this, so I skipped this lesson



https://www.udemy.com/course/one-week-python/learn/lecture/31596894#overview

Lambdas: Anonymous 1 Line Functions

Lambdas can accept any number of params and return any valid expression



Syntax

```
# Single parameter lambda
double_it = lambda a: a*2
result1 = double_it(100) # 200
# Multi-parameter lambda
add_both = lambda a, b: a + b
result2 = add_both(5, 10) # 15
```

Compare to Regular Methods

The following 2 methods are equivalent

- Keep in mind that normally when using lamdas, we don't assign them to named variables
- For the sake of this demonstration, we save a lambda to show how regular methods accomplish the exact same thing

```
def add1(num):
    return num + 1
add1_lambda = lambda num: num + 1
result1 = add1(5) #6
result2 = add1_lambda(14) # 15
```

Pass Methods as Arguments using Lambdas

Lambdas are most useful when you want to pass a method as an argument to another method (ex. sort, map, filter methods in Python)
You could define a function normally - but that takes more lines of code

REDUNDANT METHOD DEFINITION

```
fast = ["KFC", "DQ", "Royal Chinese Cuisine", "Popeyes"]
def wordLength(word):
    return len(word) # we only need to use this function 1
fast.sort(key=wordLength)
print(fast) # ['DQ', 'KFC', 'Popeyes', 'Royal Chinese Cuisi
```

MORE CLEAN

```
fast = ["KFC", "DQ", "Royal Chinese Cuisine", "Popeyes"]
fast.sort(key=lambda word: len(word))
print(fast) # ['DQ', 'KFC', 'Popeyes', 'Royal Chinese Cuisi
```

Scope

SCOPE DEFINITION
The location where a variable is initially defined will determine where it can be referenced/used

Scopes Overview

These scopes are work the same way in Python as they do in JS

Scope Table

Oftentimes we think about Local and Enclosing scope in the same vein
Declare a variable inside a function, use it on the same level or ones nested deeper - as long as they are below the declaration

OFFICIAL SCOPES

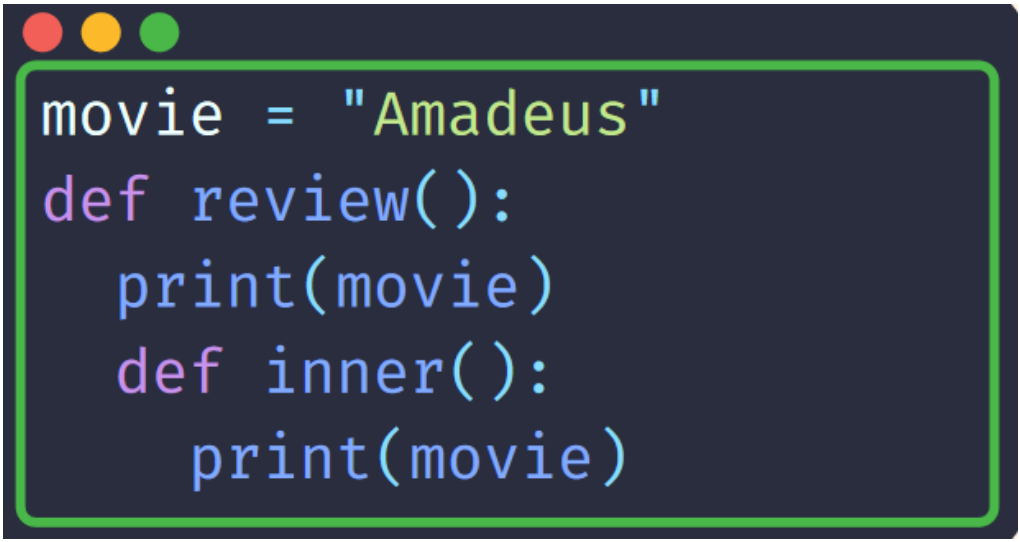
Built in

Python has many built-in objects and methods that we can access anywhere in a Python script

```
str(1.99) # "1.99" (this casting method can be used anywher
```

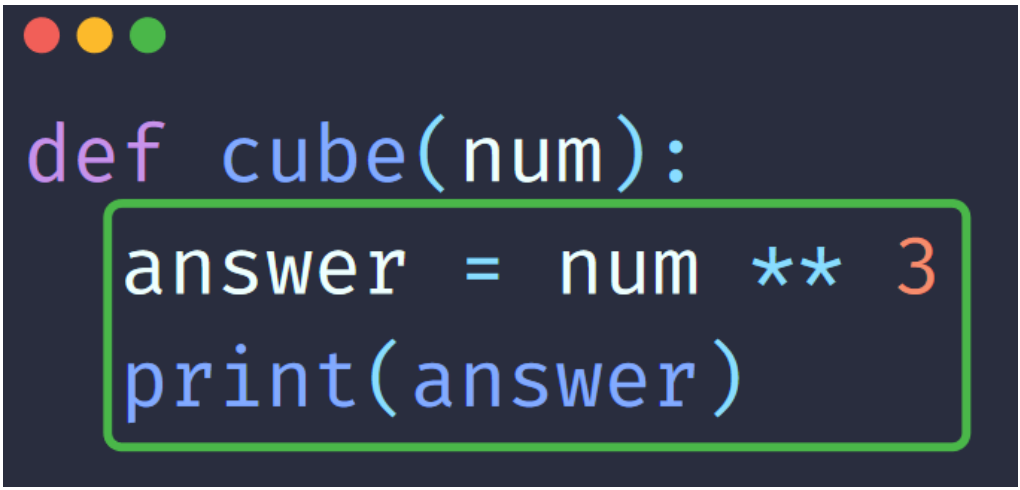
Global

Variables declared outside of functions are in the global scope.
All functions have read access to them
Editing them is not as simple however - you'll need the global or nonlocal keyword for that



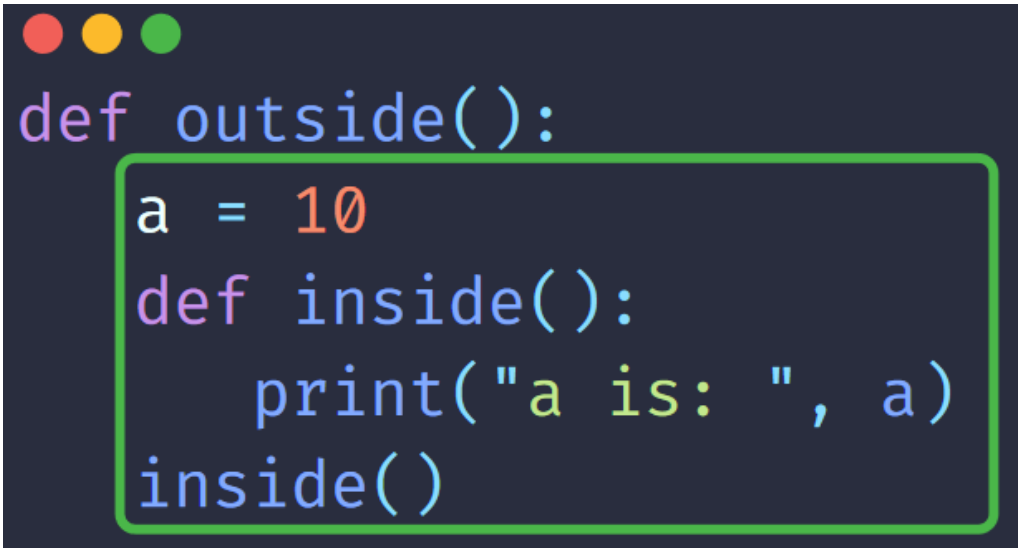
Local

Variables defined in a function are scoped to that function.
They are not available outside that function



Enclosing

Nested "inner" functions have access to variables declared in outer parent functions



Burst Scope: In a for Loop

If a variable is declared inside a for loop, it can be referenced... anywhere on that loop's level - even outside of the loop itself

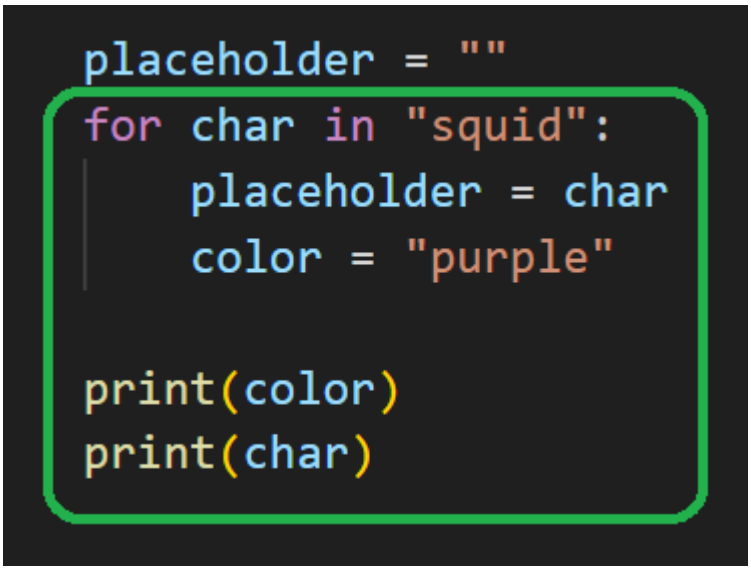
> DIFFERENCE COMPARED TO JAVASCRIPT
> If a variable is declared as part of a for loop (either in the iterator or on the indented code block)
> It can be referenced on that loop's level, even outside of the loop itself (as long as it's below the declaration)
>
> LIMITATION
> Variables must be declared before they are referenced
> In practice that just means the declaration will be on a line above where a variable is called

Example 1: For loop on Global Scope

For loop declared on the global scope
The `char` and `color` variables are defined as part of it, and are available...

- Inside the loop
- Outside the loop on the same level where that loop is declared

```
placeholder = ""
# print(char) # would cause an error if commented in (befor
for char in "squid": # char is declared on the scope of the
    placeholder = char
    color = "purple" # variable defined in a function, but
print(color) # purple  (color is referenced outside of the
print(char) # d    (char is referenced outside of the loop
```



Example 2: For Loop on Nested Function Scope

For loop declared in the demoMethod function scope
The `char` and `exclaim` variables are defined as part of it, and are available...

- Inside the for loop
- Inside the demoMethod method scope

```
def demoMethod():
    for char in "tails":
        exclaim = char + "!"
    #Both char and exclaim are available on the level the f
    # In this case, that's the demoMethod function scope
    return { "char_variable": char, "exclaim_variable": exc
save = demoMethod() # {'char_variable': 's', 'exclaim_varia
print(char) # causes SyntaxError
print(exclaim) # causes SyntaxError
```

```
def demoMethod():
    for char in "tails":
        exclaim = char + "!"

    return { "char_variable": char, "exclaim_variable": exclaim }

save = demoMethod()
```

Example 3: Nested for Loops

If you have nested for loops and a variable is declared inside of one of them The variable can be referenced on the level of the outermost one

```
for i in range(2):
    for j in range(2):
        for k in range(2):
            print(f"i={i}, j={j}, k={k}")
            test_var = 5
```

```
print(test_var)  # 5
```

```
for i in range(2):
    for j in range(2):
        for k in range(2):
            print(f"i={i}, j={j}, k={k}")
            test_var = 5

print(test_var)
```

Burst Scope: Inside Conditional Blocks

This special kind of scope works exactly the same as for loop scopes

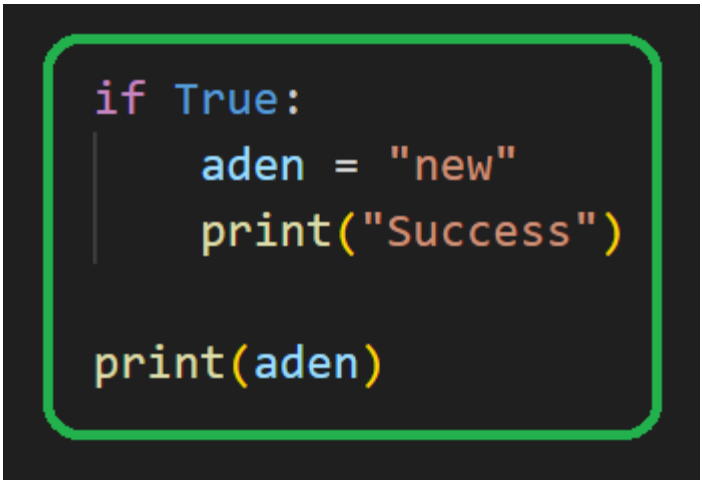
> DIFFERENCE COMPARED TO JAVASCRIPT
> If a variable is declared as part of a conditional block (if, elif, else)
> It can be referenced on the level where that conditional block was defined, as long as the log
>
> LIMITATIONS
>
> 1. The script logic must reach the part where the variable is declared for it to be referenced outside of the block]
>
> 2. Variables must be declared before they are referenced
>   In practice that just means the declaration will be on a line above where a variable is called

Example 1: In an if block on the Global Scope

If block declared on the global scope
The `aden` variable is defined as part of it, and is available...

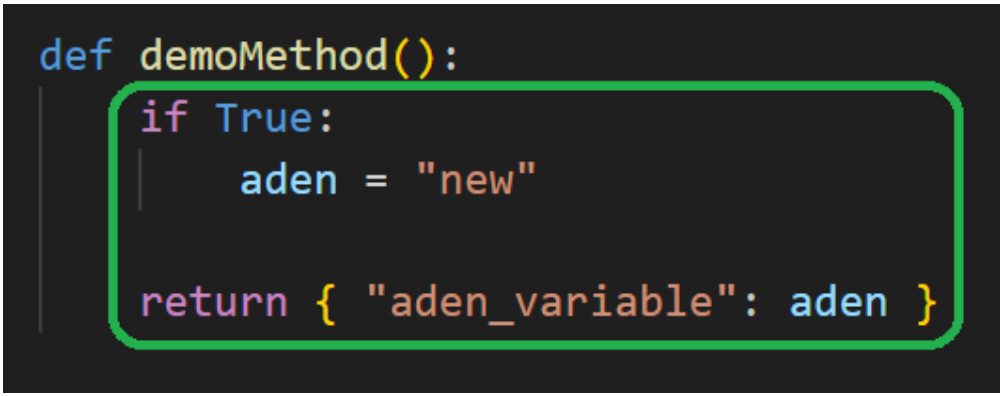
- Inside the if block
- Outside the if block on the same level where that block is declared

```
if True:
    aden = "new"
    print("Success")
print(aden) # new
```



Example 2: In an if block in a function scope

```
def demoMethod():
    if True:
        aden = "new"
    # aden available on the level the if block was declared
    # In this case, that's the demoMethod function scope
    return { "aden_variable": aden }
save = demoMethod() # {'aden_variable': 'new'}
print(aden) # causes SyntaxError
```



Example 3: If block inside Nested ifs

```
x = 10
y = 5
if True:
    if True:
        if x < y:
            aden = "new" # variable is only declared if log
        else:
            aden = "old" # variable is only declared if log
print(aden) # old
```

```
x = 10
y = 5

if True:
    if True:
        if x < y:
            aden = "new"
        else:
            aden = "old"

print(aden) # old
```

Demonstrating Special Scope Limitations

Variable Must be Declared before Use

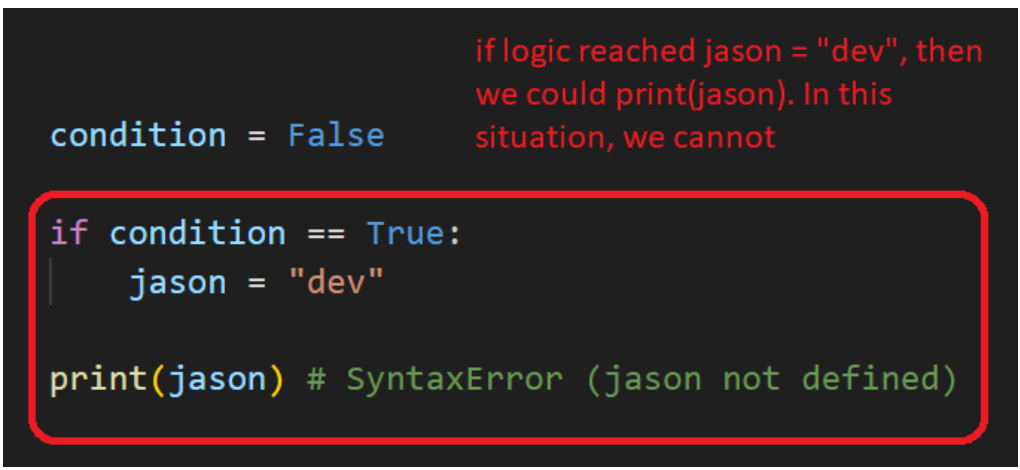
```
print(jason) # SyntaxError
```

```
if True:
    jason = "dev"

print(jason) # "dev" (if not for error)
```

Logic must Reach Variable for it to be Referenced

If the script logic does not reach where a variable is defined, it's as if the variable was never declared at all

```
condition = False
if condition == True:
    jason = "dev"
print(jason) # SyntaxError (jason not defined)
```

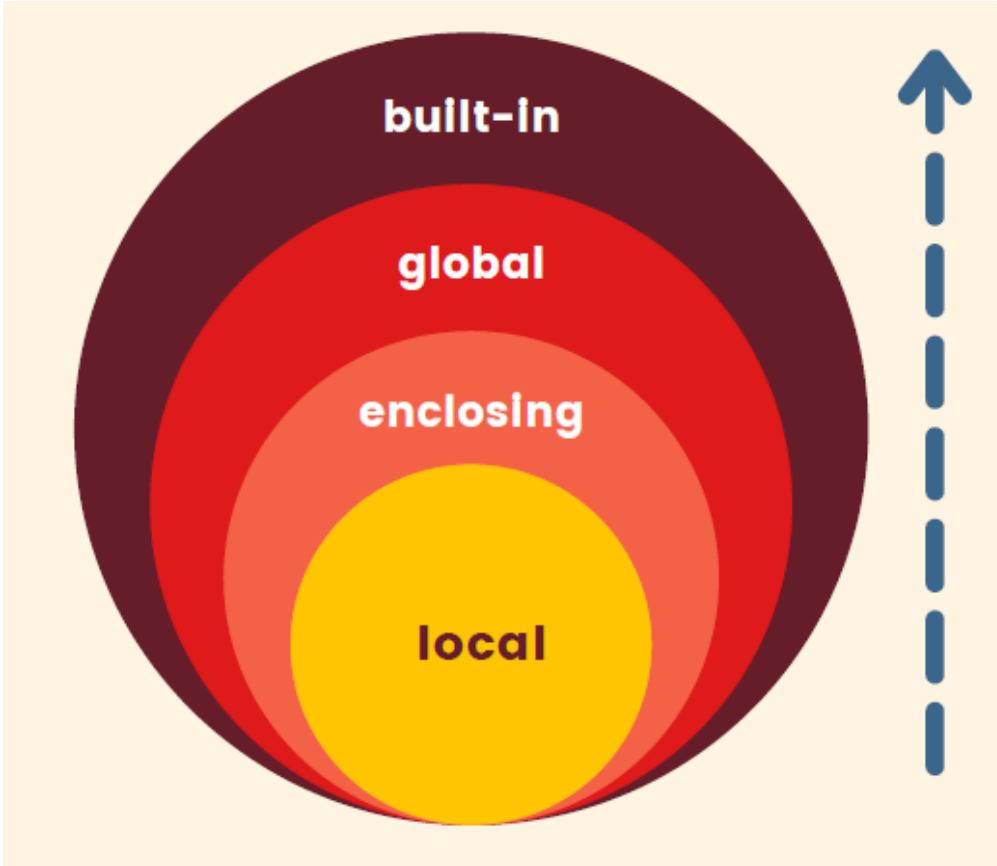

if logic reached jason = "dev", then we could print(jason). In this situation, we cannot
condition = False
if condition == True:
jason = "dev"
print(jason) # SyntaxError (jason not defined)

Scope Precedence: Diff b/t Python & JS Variable Scopes

PRE-REQUISITE:
Read "scopes Overview" lesson in this chapter before this one

SP Rules + Diagram

- You can declare variables with the same name in different scopes
- When you reference a variable, Python looks in the current scope first
- If Python cannot find that variable in the current scope, it will then look on a higher level next... etc



Examples

Example 1: (highlights how different scope is in Python vs JS)

```
a = 50 # variable "a" exists in the global scope
def demo():
    a = 100 # defines a new variable named "a" in a local s
demo()
print(a) # PRINTS: 50
# REASON: Last line prints variable "a" in the global scope
```

Example 2:

```
def demo():
    b =  100 # declare variable in local scope
a = demo()
print(b) # NameError: name 'b' is not defined
# REASON:
# b is declared on a local scope (inside the function)
# we attempt to print it on the global scope, where variabl
# variable b is not found in the built in scope (the only l
```

Example 3:

```
temperature = 78
def get_weather():
    temperature = 90 # declare new temp variable in local s
    print(temperature) # Python finds temp variable in curr
get_weather() # PRINTS 90
```

global Keyword

How JS Scope Differs from Python's

Scope Precedence makes the way Python handles scope radically different to JS
Redefining global variables inside of functions works very simply in JS while it does not in Python

JS Snippet where we redefine a variable value inside a function

```
let a = 50;
const redef = function () {
  a = 100;
};
redef();
console.log(a); // PRINTS: 100
```

Python snippet where we try to do the same thing but fail to- scope Precedence does not allow it

```
a = 50
def demo():
    a = 100 # this does not redefine the variable value (th
demo()
print(a) # PRINTS: 50
```

PYTHON ALTERNATIVE #1 (does not require global keyword)
If we want to redefine a variable value, do it in the same scope where it was declared

```
a = 50
def demo():
    return 100
a = demo() # this redefines the value of a
print(a) # PRINTS: 100
```

Global Keyword

Python offers a way for us to define/redefine globally-scoped variables at non-global parts of our script

```
score = 10
def double_points():
    global score # tells python to use the global "score" v
    score = score * 2
print(score) # PRINTS: 10 (the function has not been called
double_points()
print(score) # PRINTS: 20 (the function redefines the value
```

Using global is a Bad Practice

Using Global variables is considered a bad practice for several reasons - so use them sparingly

Code Maintainability:

- Global variables can be accessed and modified from anywhere in your code, making it difficult to track down where changes are happening.
- This can lead to unexpected behavior and make it challenging to maintain and update your code.

Code Reusability:

- Global variables can limit the reusability of your code.
- If a function relies on global variables, it becomes less self-contained and harder to reuse in different contexts.

Debugging Complexity:
Debugging becomes more challenging when global variables are involved because you need to consider the entire program's state when trying to find the source of a problem.

Testing Difficulty:
Writing unit tests for functions that use global variables can be complicated because you may need to set up and tear down global state for each test case, which can be error-prone.

Code Isolation:
Global variables break the principle of encapsulation, where data should be kept within the scope of the functions that use them

Interesting Error: Variable Referenced Before Assignment

2:00 mark
https://www.udemy.com/course/one-week-python/learn/lecture/31340864#notes

nonlocal keyword

The nonlocal keyword is similar to global, except you use it in scenarios where...
You want to edit non-global variables declared in a higher scope (the closest one where you use it)

Rules

- Can only be used in nested functions (functions defined inside other functions)
- Cannot create new variables - the variable must exist in an enclosing scope
- Cannot reach the global scope

Example

```
def outer():
    count = 0  # Variable in outer function's scope

    def inner():
        nonlocal count  # Tells Python we want to modify ou
        count += 1      # Modifies the outer function's var

    inner()
    return count
```

Without nonlocal, we'd get the following error:

```
UnboundLocalError: cannot access local variable 'count' where it is not associated with a value
```

Decorators: Syntactic Sugar for Wrapper Upgrade Functions

Higher Order Functions

There are 2 types of higher order functions

1. Function that accepts another function as an argument
2. Function that returns another function

Function that accepts another function as an argument

`higher_order_fn` is a function
It accepts another function `fn` as an argument

```
def higher_order_fn(dict, fn, key):
    return fn(dict, key)
dict = {"a": 1, "b": 2}
def drill(dict, key):
    return dict.get(key)
output = higher_order_fn(dict, drill, "b")
print(output)  # 2
```

Function that returns another function

`higher_order_fn` returns a different function depending on the parameter

supplied to it

```
import random
import string
def higher_order_fn(requirement):
    # Define 3 functions that may be returned by higher_ord
    def randomNum():
        return random.randint(1, 100)
    def randomSymbol():
        symbols = "!@#$%^&*()_-+=<>?/"
        return random.choice(symbols)
    def randomLetter():
        return random.choice(string.ascii_letters)
 # Return a function that depends on the parameter supplied
    if requirement == "number":
        return randomNum
    elif requirement == "letter":
        return randomLetter
    else:
        return randomSymbol
```

Intro to Decorators

What are Decorators?

Decorators are higher order functions

- They wrap around other functions to enhance their behavior
- Decorators will accept a function and return an enhanced version of it

Syntax Explained

The @ symbol is syntactic sugar

- Allows us to skip the cumbersome task of wrapping a function and returning enhanced versions of it
- When `@decorator` is placed directly above a method, that method will be enhanced whenever its name is called

```
def be_polite(fn):
    def wrapper():
        print("What a pleasure to meet you!")
        fn()
        print("Have a great day!")
    return wrapper

def greet():
    print("My name is Colt.")

greet = be_polite(greet)
# we are decorating our function
# with politeness!
```

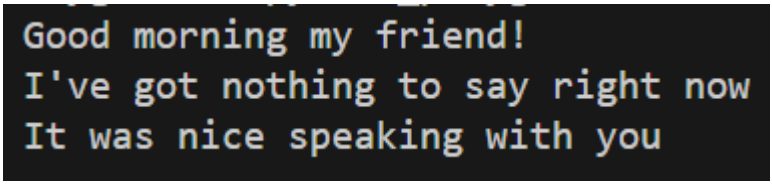
```
def be_polite(fn):
    def wrapper():
        print("What a pleasure to meet you!")
        fn()
        print("Have a great day!")
    return wrapper

@be_polite
def greet():
    print("My name is Matt.")

# we don't need to set
# greet = be_polite(greet)
```

On Functions with 1 arg

Create a decorator that takes a function that logs a message
Make it more polite by logging a message before and after it

With decorators

```
def upgrade(fn):
    def wrapper():
        print("Good morning my friend!")
        fn()
        print("It was nice speaking with you")
    return wrapper  # return an enhanced version of fn
@upgrade
def message():
    print("I've got nothing to say right now")
message()
```



Without decorators
We'd need to create a new variable `upgraded_message` which holds the enhanced version of message

```
def upgrade(fn):
    def wrapper():
        print("Good morning my friend!")
        fn()
        print("It was nice speaking with you")
    return wrapper  # return an enhanced version of fn
def message():
    print("I've got nothing to say right now")
upgraded_message = upgrade(message)
upgraded_message()
```

On Functions w/ Multiple Args

ISSUE WITH MULTIPLE ARGUMENTS
Decorators raise exceptions when the wrapper function has a different number of args than the function being wrapped

```
def capitalize(fn):
    def wrapper(txt):
        return fn(txt).upper()
    return wrapper # return enhanced fn where returned text
@capitalize
def order(main, side):
    return f"I'd like to order the {main}, with a side of {
print(order("burger", "fries")) # EXCEPTION THROWN
```

```
Traceback (most recent call last):
  File "c:\Users\jason.bustamante\Desktop\Playgrounds\python_playground\starter.py", line 15, in <module>
    print(order("Burgers", "fries"))
          ^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: capitalize.<locals>.wrapper() takes 1 positional argument but 2 were given
```

SOLUTION:
Make the wrapper function accept an unspecified number of args using `*args` and `kwargs` at the same time

```
def capitalize(fn):
    def wrapper(*args, **kwargs):
        return fn(*args, **kwargs).upper()
    return wrapper
@capitalize
def greet(name):
    return f"Hi, my name is {name}"
@capitalize
def order(main, side):
    return f"I'd like to order the {main}, with a side of {
print(greet("Mason")) # HI, MY NAME IS MASON
print(order("burger", "fries")) # I'D LIKE TO ORDER THE BUR
```

Use Case: Get Function Speed

We're creating a decorator function that lets us get the execution time of the methods we wrap it around

```
from time import time
def calc_speed(fn):
    def wrapper(*args, **kwargs):
        start_time = time()
        result = fn(*args, **kwargs)
        elapsed_time = round(time() - start_time, 2)
        print(f"Seconds: {elapsed_time}")
        return result
    return wrapper  # return enhanced fn where returned tex
# TEST IT OUT
@calc_speed
def sum_nums():
    total = 0
    for n in range(10000000):
        total += n
    return total
result = sum_nums() # OUTPUT: 49999995000000   LOGGED: 2.15
```

Unpacking: Python Destructuring

Lists, Tuples, & Sets

FUN FACT
Variable names can be anything while unpacking - so there's no need for renaming variables

Lists

```
my_list = [1, 2, 3]
a, b, c = my_list
print(a)  # 1
print(b)  # 2
print(c)  # 3
```

Sets

```
nums =  {"pear", "apple", "melon"}
x, y, z = nums
print(x) # pear
print(y) # apple
print(z) # melon
```

Tuples

```
my_tuple = (4, 5, 6)
x, y, z = my_tuple
print(x)  # 4
print(y)  # 5
print(z)  # 6
```

Dictionaries

FUN FACT
Variable names can be anything while unpacking - so there's no need for renaming variables

Not like JS

Unpacking dictionaries in Python objects works differently than destructuring objects in JS
You cannot reference dictionary keys to extract their value

Javascript

```
const obj = { a: 1, b: 3 };
const { b } = obj;
console.log(b); // 3
```

Python (wtf is going on?)

```
obj = {'a': 1, 'b': 3}
b,a = obj
print(b) # a
print(a) # b
```

How Unpacking Dictionaries Works

OPTION 1
Import a built in function in Python - `itemgetter`

```
from operator import itemgetter
obj = {'a': 1, 'b': 3}
var1, var2 = itemgetter('a','b')(obj)
print(var1) # 1
print(var2) # 3
# WARNING:
# A keyError occurs if you try to use itemgetter using a ke
```

```
from operator import itemgetter
obj = {'a': 1, 'b': 3}
var1, var2 = itemgetter('b','a')(obj)
print(var1) # 3
print(var2) # 1
```

OPTION 2
Multiple bracket notations per line ()

```
dict = {'a': 1, 'b': 3, 'c': 5}
var1, var2 = dict['a'], dict['b']
print(var1) # 1
print(var2) # 3
```

Assign Vars to Remaining Items

If you unpack less than the amount of items in a sequential data structure, you can place the remaining ones in a list using `*`
This does not work on dictionaries

Lists

```
my_list = [1, 2, 3]
a, *remaining = my_list
print(a)  # 1
print(remaining)  # [2, 3]
```

Sets

```
my_set = {1, 2, 3}
a, *remaining = my_set
print(a)  # 1
print(remaining)  # [2, 3]
```

Tuples

```
my_tuple = (1, 2, 3)
a, *remaining = my_tuple
print(a)  # 1
print(remaining)  # [2, 3]
```

JSON

json.dumps() : Turn Dict into String

This is Python's version of `stringify` from JS

```
import json
v1 = {"documentType": "purchase_order"}
v2 = json.dumps(v1)
print(v2)     # {"documentType": "purchase_order"}
print(type(v2))   # <class 'str'>
```

= = = = = = = CONDITIONALS = = = = = = =

Booleans & Truthy/Falsy

Booleans

Booleans are a data type in Python that have 2 possible values: `True` and `False`
Note that the capital letters to start are important

```
a = True
b = False
```

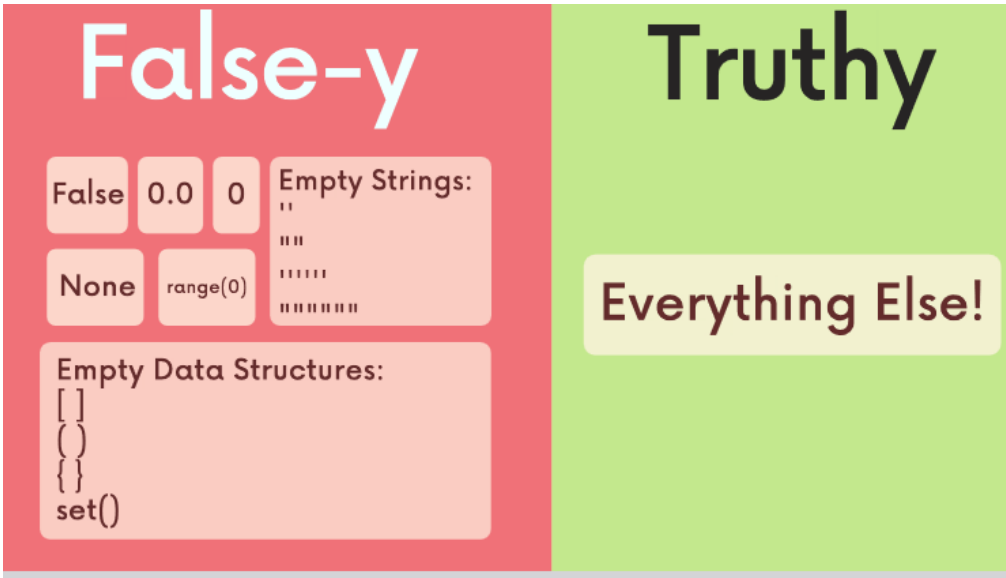
Many expressions can evaluate to True or False

```
"565543".isnumeric() # True
5 > 10 # False
```

Truthy & Falsy

Every value or expression in Python boils down to a Truthy or Falsy in Python
This is useful for writing conditionals and to save time type casting

- Though you can type cast values into Booleans with `bool()` if you want
- It's more explicit to have True/False values over Truthy/Falsy, but it's up to you and situation-dependent



MAJOR DIFFERENCE COMPARED TO JS:
Empty data structures like arrays or objects would not be considered a falsy in JS
In Python they are

bool() : Type Cast a Value to Boolean

This is a snippet from the type casting lesson

```
list1 = [1, 2, 3] # a Truthy
is_list1_filled = bool(list)  # True
list2 = [] # empty data structures are considered Falsys in
is_list2_filled = bool(list2)  # False
```

Conditionals

if elif else

```
if condition:
 execute code if above condition is true
elif condition:
 execute code if above condition is true
else:
 execute code if none of the conditions were evaluated to t
```

Demo

```
age = -20
if age >= 18:
    print("User is an adult")
elif age >= 13:
    print("User is a teenager")
elif age >= 0:
    print("User is an child")
else:
    print("Invalid age")
print("I always run since I'm not indented")
```

```
User is a teenager
I always run since I'm not indented
```

Indents

Python does not use curly braces to define scope

- It relies on indentation to decide what code to run if a condition is met (see above example)
- The typical indentation size is 4 spaces, but you can use a different amount as long as you stay consistent
  Can even be 1 space - but it's most common to use 4 or 2

Watch on 2X speed to see importance of indentation
https://www.udemy.com/course/one-week-python/learn/lecture/31122882#overview

Logic Flow is the same as JS

This is a syntax change, and nothing more

- Free to use `if` on its own
- Cannot use `elif` or `else` without an `if`
- The first condition that is met will have it's indented code block ran
  Any later `elif` or `else` blocks will not run afterward

Nested ifs

Nested ifs are just a matter of using additional ifs indented further to the right
Be sure to keep your indenting consistent

Demo

```
age = 27
gender = "M"
wearing_costume = True
if age >= 18:
    if gender == "F" or wearing_costume:
        print("You get a club event discount")
    else:
        print("You get standard club event pricing")
else:
    print("You may not enter the club")
```

```
You get a club event discount
```

Careful with Indentation

Nested ifs in particular take on different meanings depending on the indentation

Snippet 1:

```
fave_color = "green"
height = 5
if fave_color == 'purple':
    print("How royal!")
    if height < 6:
        print("Short king")
```

(This code snippet has no output)

Snippet 2: (changed indentation)

```
fave_color = "red"
height = 5
if fave_color == 'purple':
    print("How royal!")
if height < 6:
    print("Short king")
```

```
Short king
```

Replace with logical Operators when Possible

Although nested ifs aren't necessarily bad, you don't want your logic too nested
Replace nested ifs with logical operator usage when possible (focus of another lesson)

```
# Nested ifs (a bit messy)
if age >= 18:
    if gender == "F" or wearing_costume:
        print("You get a club event discount")
    else:
        print("You get standard club event pricing")
else:
    print("You may not enter the club")
# Cleaner with logical and
if age >= 18 and (gender == "F" or wearing_costume):
    print("You get a club event discount")
elif age >= 18:
```

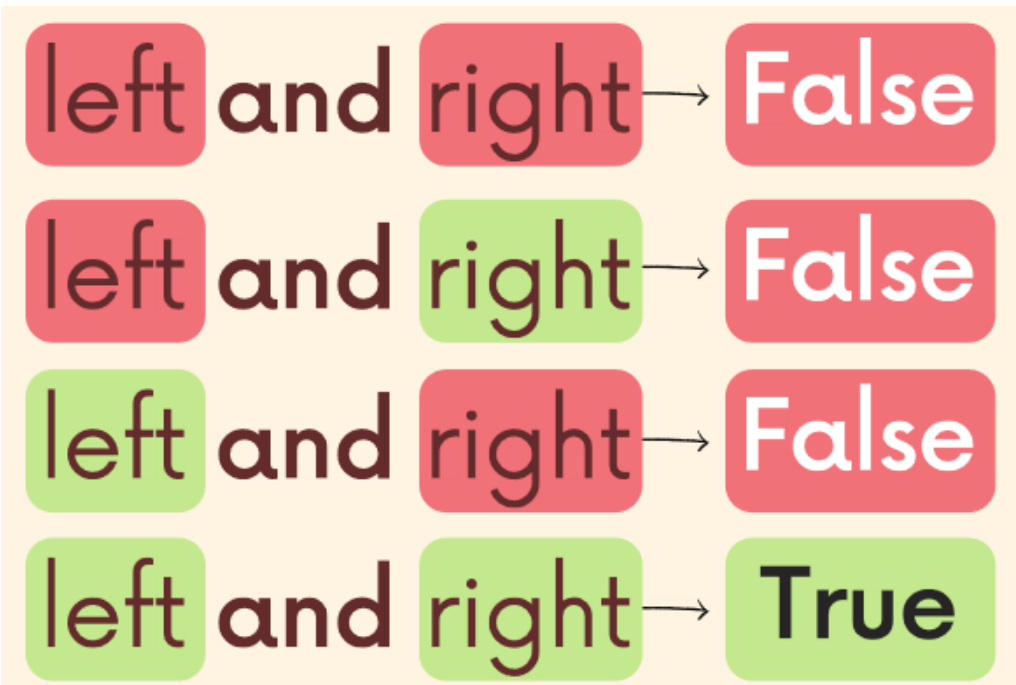
```
    print("You get standard club event pricing")
else:
    print("You may not enter the club")
```

Logical Operators

and

How Operator works

Evaluates to true if both the left/right sides evaluate to true



Example

SCENARIO:
2 people visit a rollercoaster ride - and both need to be at least 60 inches tall to ride

Both sides are truthy

```
person_1_height = 62
person_2_height = 70
can_ride = person_1_height >= 60 and person_2_height >= 60
```

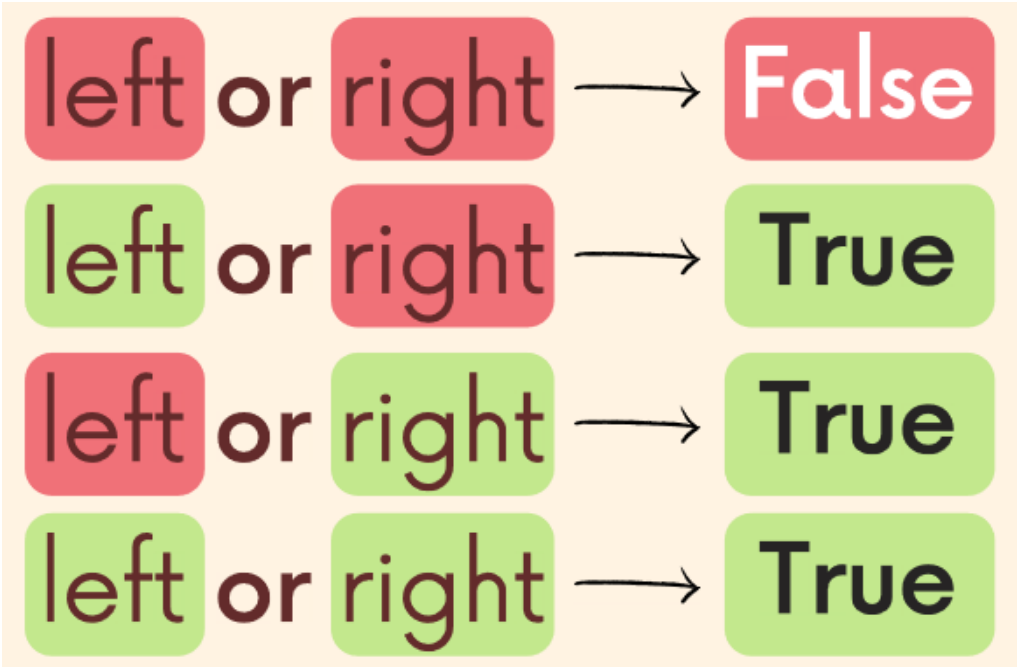
Both sides are not truthy (either 1 of them is, or neither of them are)

```
person_1_height = 40
person_2_height = 70
can_ride = person_1_height >= 60 and person_2_height >= 60
```

or

How Operator Works

Evaluates to true if 1 or both sides evaluate to True



Example

SCENARIO:

2 people go to the liquor store - at least 1 person needs to be of age to buy some

Right side is Truthy

```
person_1_age = 10
person_2_age = 40
can_buy_alcohol = person_1_age >= 21 or person_2_age >= 21
```

Neither Side is Truthy

```
person_1_age = 10
person_2_age = 2
can_buy_alcohol = person_1_age >= 21 or person_2_age >= 21
```

not

The `not` operator will negate the expression that comes after it

- Turns True/Truthy values False
- Turns False/Falsy values to True

```
a = 0  # Falsy
print(not a)  # True
b = [1, 2, 3]  # Truthy
print(not b)  # False
```

Ternary Operator Expressions

Expression Syntax

```
ternaryExpression = value1 if condition_expression else value2
```

Demo

If age is over 18, set can_drink to True
Otherwise set can_drink to False

```
age = 19
can_drink = True if age >= 18 else False
print(can_drink) # True
```

Set `team` equal to `optionA` if a random number you generate equals 0
Otherwise, set `result` to `optionB`

```
import random
random_integer = random.randint(0, 1)  # Generates 0 or 1 r
optionA = "Ascendent Frontier"
optionB = "Band of Brothers"
team = optionA if random_integer else optionB # AF if rando
print(team, random_integer)
# Possible Printed Outcomes
# Ascendent Frontier 1
# Band of Brothers 0
```

Conditional Increment

Increment counter by 1 if condition is true
Leave it as is otherwise

```
unique_s_chars += 1 if s_freq[char] == 1 else 0
# Encapsulate the ternary operator expression in brackets
unique_s_chars += (1 if s_freq[char] == 1 else 0)
```

Use Case: Eliminate Nested ifs

Avoid deeply nested ifs by making use of logical operators

Nested if Example

```
if age >= 18:
    if gender == "F" or wearing_costume:
        print("You get a club event discount")
    else:
        print("You get standard club event pricing")
else:
    print("You may not enter the club")
```

Use logical operators to elim nested if

```
if age >= 18 and (gender == "F" or wearing_costume):
    print("You get a club event discount")
elif age >= 18:
    print("You get standard club event pricing")
else:
    print("You may not enter the club")
```

Other Operators + Comparisons

Operators Overview

Comparison Operators

All of these comparison operators evaluate to a Boolean value

Arithmetic Operators

Assignment Operators

Assume `x = 6` for the following tables examples

All of these assignment operators act as shortcuts (aside from `=`)
Feel free to use any of the equivalent expressions

String Operators

Repeated at JUMP119

Keep in mind that `in` and `not in` work on other sequence in python aside from strings
They have a use case in lists as well for example (but we won't focus on that right here)

Identity Operators

See dedicated lesson later in this chapter

Identity Operators: "is" and "is not"

Identity operators tell you if 2 variables point to the same object in memory

- Perfect for checking if variables have Boolean and None values
  Acts like an equality operator for values like these
- Good for checking if Object types point to the same location in memory
  Ex. lists, tuples, sets, dictionaries... etc

Checking for Boolean & None

CHECKING FOR NONE VALUE
When `None` is used, it always points to the same location in memory

```
a = None
b = None
c = a
print(a is b)  # True
print(c is a)  # True
if a is None:
    # do something

if a is not None:
    # do something
```

CHECKING FOR BOOLEAN VALUES
Both True and False point to the same 2 ocations in memory regardless of where they are declared

```
a = False
b = False
c = a
print(a is b) # True
print(c is a) # True
if a is False:
    # do something

if a is not False:
    # do something
```

Object-types

is

```
a = [1, 2, 3]
b = [1, 2, 3]
c = a
print(a is b) # False, because b is a different object in m
print(c is a) # True, because c was assigned to a
print(a is c) # True, because c was assigned to a
```

is not

```
a = {'score': 1, 'shots': 2}
b = {'score': 1, 'shots': 2}
c = a
print(a is not b) # True, because a and b point to differen
print(c is not a) # False, because c and a point to the sam
```

```
print(a is not c) # False, because c and a point to the sam
```

Do not use on Numbers

Identity operators can work on certain numbers since Python caches small numbers beneath 256

- Not recommended since the behavior can be unpredictable at times
- Use equality operators like `==` or `!=` instead

```
a = 100
b = 100
print(a is b)   # True, since both refer to the same cached
x = 1000
y = 1000
print(x is y)   # False, since these are different objects i
```

List of Data Types they Work on

These are all the types the `is` and `is not` operators can be used to compare

Order Of Operators: aka Operator Precedence

What an Order of Operations is

It gets confusing when several operators are used on multiple operands

- Python has a specific order it will apply operators in
- Use brackets to bunch operations together so you don't need to memorize the order

Order of Math Operators

1. Parentheses `()`
2. Exponents `**`
3. Multiplication `*` and Division `/`
4. Addition `+` and Subtraction `-`

Notice how multiplication is applied first to make the expression `10 - 8`

```
a = 5 * 2 - 8
print(a) # 2
```

We can use brackets to change the order

```
a = 5 * (2 - 8)
print(a) # -30
```

Order of All Operators

We haven't covered this topic yet, but the operators `not, and , or` also have an order as well
They are part of the same order the math operators are in

- Personally I just use brackets to help make logical expressions clear
- If you absolutely need a guide to understand the order of an expression, check here:
  https://www.scaler.com/topics/operator-precedence-in-python/

Comparing Values with Different Data Types

How Python usually Acts

Python is not lenient when you compare values with different data types

```
a = 4.0 == "4"
print(a) # False
```

SOLUTION:
Use type casting (focus of next lesson) to coerce values into different types

Exception: Comparing Floats to Integers

Python lets you strictly compare integers to floats intuitively

```
a = 4.0 == 4
print(a) # True
```

< Unicode Comparisons on Strings >

Normally we don't use certain comparison operators on strings

But if we do, it involves unicode:
https://www.udemy.com/course/one-week-python/learn/lecture/31018664#overview

```
strange_comparison = "100" < "9" # True
```

= = = = = = = = = = LOOPS

= = = = = = = = = =

for

Looping over all Iterables

There are many data structures in Python that count as iterables - and you can for loop through them all

Strings

```
phrase = "Stop it"
for char in phrase:
  print(char)
```

```
S
t
o
p

i
t
```

Lists

```
my_list = [1, 2, 3, 4, 5]
for item in my_list:
    print(item)
```

Tuples

```
my_tuple = (1, 2, 3, 4, 5)
for item in my_tuple:
    print(item)
```

Sets

```
my_set = {1, 2, 3, 4, 5}
for element in my_set:
    print(element)
```

Dictionaries

```
my_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}
for key in my_dict:
    print(key, my_dict[key])
```

range()

Creates a range of numbers
Index range: Beginning inclusive, end exclusive

```
range(start, stop, step)
```

Hardcoded Loop Length

Create range from 0 to 3

```
for n in range(4):
  print(n)
```

```
0
1
2
3
```

Create range from -5 to 0

```
for n in range(-5, 1):
  print(n)
```

```
-5
-4
-3
-2
-1
0
```

Create a range from -10 to 10 with increments of 2

```
for n in range(-10, 11, 2):
  print(n)
```

```
-10
-8
-6
-4
-2
0
2
4
6
8
10
```

Create a range from 10 to 4 (count backwards)

- Starting index 10 is included
- -1 step means we count backwards
- End index of 3 means the last index included will be 2

```
for n in range(10, 3, -1):
  print(n)
```

```
10
9
8
7
6
5
4
```

Dynamic Loop Length

```
my_list = [0, 1, 3, 5]
for i in range(len(my_list)):
    print(my_list[i])
```

```
0
1
3
5
```

Range without Accessing i

Sometimes you just want to use a range without caring about the index
In cases like these, use an underscore _

```
for _ in range(self.length - 2):
```

Controlling range()

How to Memorize Args

The range method has 3 arguments (some optional and some required)
To be safe, define all 3 each time and remember what they are

```
for _ in range (start_index, end_index, step):
    # start_index is bound inclusive
```

```
    # end_index is bound exclusive (up to but not including
    pass
```

Exercises for Controlling Range

Count backwards from 5 to 0

```
for num in range(5, -1, -1):
    print(num)
# 5 4 3 2 1 0
```

Count from 0 to 10 (skip all even numbers)

```
for num in range(1, 11, 2):
    print(num)
# 1 3 5 7 9
```

Nested for Loops

You can nest for and while loops within each other if needed

```
my_dictionary = {
    "nFuego": [100, 90, 42],
    "xPyra": [0, 200, 44, 9]
}
for key in my_dictionary:
    list = my_dictionary[key]
    for entry in list:
        print(entry)
```

```
100
90
42
0
200
44
9
```

break: End all Iterations

The break keyword will stop a loop from continuing
Works in both for and while loops

```
phrase = "abcde"
for char in phrase:
    if char == "d":
        break  # skip iteration if the character is 'b'
    print(char)
```

```
a
b
c
```

continue: Skip Current Iteration

The `continue` keyword lets you skip the rest of current iteration from executing - then moves onto the next one
Works in both while and for loops

```
phrase = "abcde"
for char in phrase:
  if char == "b":
     continue # skip iteration if the character is 'b'
  print(char)
```

Access Index Values when Looping

```
my_list = ['apple', 'banana', 'cherry']
for index, item in enumerate(my_list):
    print(index, item)
```

```
0 apple
1 banana
2 cherry
```

List of Iterables in Python

You can loop over anything considered an iterable in Python

- Lists
- Tuples
- Strings
- Sets
- Dictionaries

Looping Backwards

All the following methods are equally performant O(n)

Standard

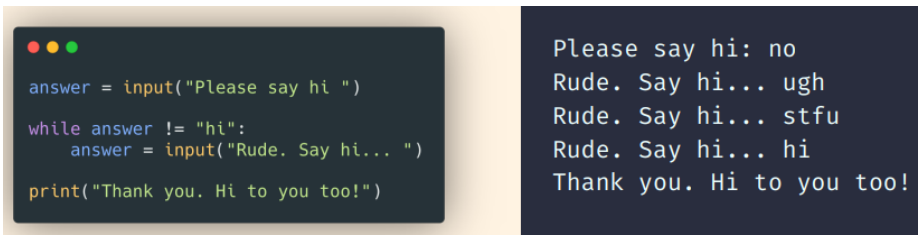
```
# Loop through a list backwards
my_list = ['a', 'b', 'c', 'd']
for item in reversed(my_list):
    print(item)  # Prints: d, c, b, a
```

When you need to Access Index

```
my_list = ['a', 'b', 'c', 'd']
for i in range(len(my_list)-1, -1, -1):
    print(my_list[i])  # Prints: d, c, b, a
```

while

While Loops


answer = input("Please say hi ")
while answer != "hi":
answer = input("Rude. Say hi... ")
print("Thank you. Hi to you too!")
Please say hi: no
Rude. Say hi... ugh
Rude. Say hi... stfu
Rude. Say hi... hi
Thank you. Hi to you too!

Standard Variant

The standard while loop iterates until the expression no longer evaluates to True

```
age = 10
while age <=18:
  print("Current age where you cannot drink: ", age)
  age += 1 # increment age variable
```

```
Current age where you cannot drink:  10
Current age where you cannot drink:  11
Current age where you cannot drink:  12
Current age where you cannot drink:  13
Current age where you cannot drink:  14
Current age where you cannot drink:  15
Current age where you cannot drink:  16
Current age where you cannot drink:  17
Current age where you cannot drink:  18
```

If Break Variant

This kind of while loops iterates forever until the if condition inside is met

```
age = 10
while True:
    if age > 18:
        break # ends the loop if the condition is met, whic
    print("Current age where you cannot drink: ", age)
    age += 1 # increment age variable
```

(results are the same since this snippet is an equivalent of the first)

While Guards: Avoid Infinite Loops

Risk of While Loops

- While loops require a their condition to eventually not be met in order to stop
- If that condition never fails to be met, your script runs to infinity and you risk crashing your PC

Solution: While Guards

- While guards involves you writing an extra condition that terminates the while loop
- Make sure this condition is guaranteed to be met by employing a counter variable

EXAMPLE:

```
lousy__while_condition = True  # mimics a condition that ma
counter = 0 # prevents while loop from running forever
while lousy__while_condition:
    if counter >= 100: # this condition will be met if the
        break
    print("Execute block of code", counter)
    counter += 1
```

Nested while loops

You can nest for and while loops together if needed

```
my_dictionary = {
    "nFuego": [100, 90, 42],
    "xPyra": [0, 200, 44, 9]
}
keys = list(my_dictionary.keys())
i = 0
while i < len(keys):
    key = keys[i]
    my_list = my_dictionary[key]
    j = 0
    while j < len(my_list):
        entry = my_list[j]
        print(entry)
        j += 1
    i += 1
```

break: End all Iterations

The break keyword will stop a loop from continuing
Works in both for and while loops

```
phrase = "abcde"
index = 0
while index < len(phrase):
    char = phrase[index]
    if char == 'd':
        index += 1  # must still iterate index to avoid inf
        break  # end iterations if the character is 'd'
    print(char)
    index += 1
```

```
a
b
c
```

continue: Skip Current Iteration

The `continue` keyword lets you skip the rest of current iteration from executing - then moves onto the next one
Works in both while and for loops

```
phrase = "abcde"
index = 0
while index < len(phrase):
    char = phrase[index]
    if char == 'b':
        index += 1  # must still iterate index to avoid inf
        continue  # skip iteration if the character is 'b'
    print(char)
    index += 1
```

```
a
c
d
e
```

MODULES

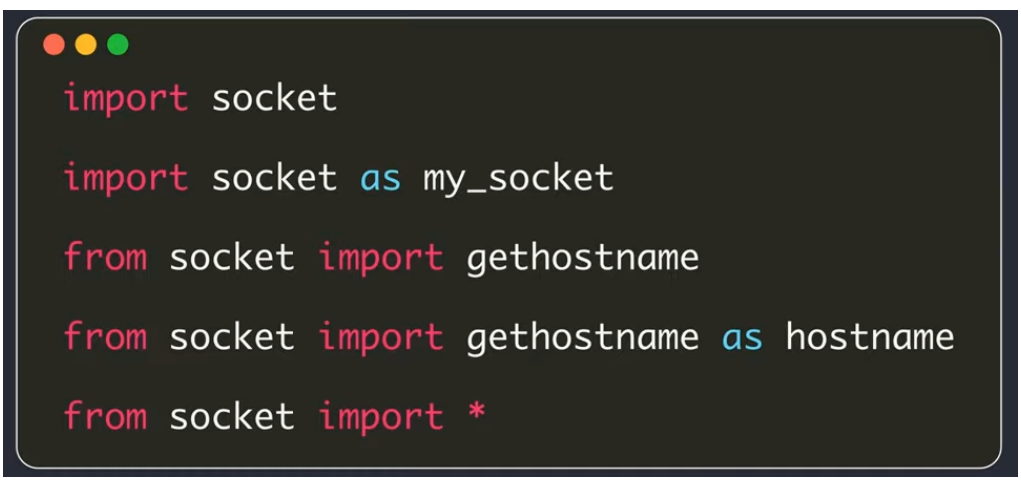
Modules Overview + Syntax for Imports

Types of Modules

Modules are Python scripts that contain code that can be reused in other files via imports (there are 3 kinds)
They help us break complex logic up into smaller manageable pieces

Import Syntax Variants

These have all been covered or used across this chapter


import socket
import socket as my_socket
from socket import gethostname
from socket import gethostname as hostname
from socket import *

Change Import Names

You can change the name used to reference imports - works with all types of modules (not just built-ins)
May help you write code that's nicer to look at

Without renaming we must use `random`.

```
import random # must import this to use random.randint()
random_integer = random.randint(1, 3)  # Generates a random
print(random_integer) # 55
```

With renaming we can use `rand`.

```
import random as rand # can now refer to random as rand ins
random_integer = rand.randint(1, 30)  # Generates a random
print(random_integer) # 55
```

Import Specific Methods from Package

If you only need 1 or 2 methods from something you import, use the special syntax that lets you specify what they are

Import everything in `math` module

```
import math
x = math.sin(1) # 0.8414709848078965
y = math.pi # 3.141592653589793
```

Import specific methods from `math` module

```
from math import pi, sin
x = sin(1) # 0.8414709848078965
y = pi # 3.141592653589793
```

import * (star): Import Everything by Name

What the Syntax Means

The `import *` expression takes every exportable thing inside the module and imports it to your script using its name
You won't need to use `module_name.method_name` in your code, just use `method_name`

```
from module_name import *
```

Demo

Default import syntax - requires you to use `math.`

```
import math
x = math.sin(1) # 0.8414709848078965
y = math.pi # 3.141592653589793
```

Import everything by name

```
from math import *
x = sin(1)  # 0.8414709848078965
y = pi  # 3.141592653589793
```

Horrible Practice to use this

TAKEAWAY FROM LESSON
Just be aware what import * does since you'll likely encounter it

WHY IT'S BAD PRACTICE

- It's tough to see what packages certain methods originate from when you use `import *`
- Stick to the default syntax or target specific method names

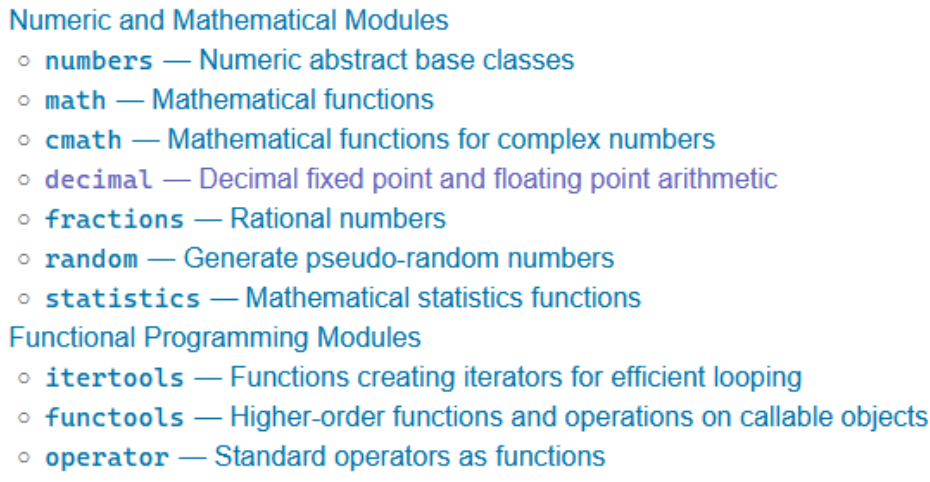
Built-in Modules Overview

List of Built-in Modules

Python's documentation contains a list of all its built-in modules
https://docs.python.org/3/library/index.html

- Use this list to find module names that may be useful, then figure out how they work with the Python docs or outside sources
(the picture below only shows a few of them)

- You can import their modules and use them in your code without any 3rd party installs


Numeric and Mathematical Modules
numbers — Numeric abstract base classes
math — Mathematical functions
cmath — Mathematical functions for complex numbers
decimal — Decimal fixed point and floating point arithmetic
fractions — Rational numbers
random — Generate pseudo-random numbers
statistics — Mathematical statistics functions
Functional Programming Modules
itertools — Functions creating iterators for efficient looping
functools — Higher-order functions and operations on callable objects
operator — Standard operators as functions

Import Built-in Module with Default Syntax

```
import random # must import this to use random.randint()
random_integer = random.randint(1, 3)  # Generates a random
print(random_integer) # 55
```

Custom Modules, Imports, Paths

Definitions

MODULE:
A Python script - usually one with variables declared on the global scope so they can be imported and used inside other Python files

PACKAGE:
Will be a directory with 1 or multiple modules that contains a single `__init__.py` file
It's essentially a collection of Python modules

Intro to Custom Modules

What gets Exported in a Python Module

Every Python file is treated as a module

- There is no need to specify what gets exported in a script
- Everything declared on the global scope of a module can be accessed as an attribute once it's imported
  That means everything declared on the root level of the file - but nothing inside of functions or other nested scopes

All Code in a Module is ran

When you import a Python module, all the code inside that script is ran.

- In the following example, "util.py has been ran" runs because we import itil.py in main.py (our deployment script)
- Even if we did not use `util.shout()`, the "util.py has been ran" would have still been logged

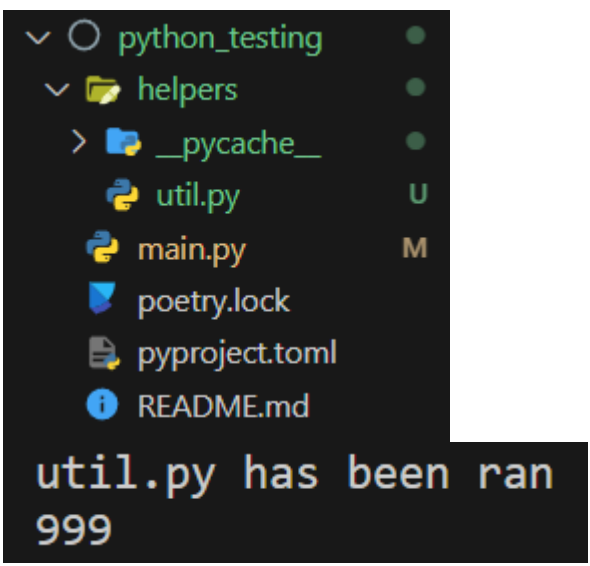
EXAMPLE:
We use a relative import with renaming (will be explored more later in the chapter)

helpers/util.py

```
print("util.py has been ran")
def shout():
    print(999)
```

main.py

```
from datetime import datetime, date
import helpers.util as util
util.shout()
```



init.py: Create Custom Packages

What this file does

The `__init__.py` file turns a directory into a package

- Import everything in the directory you'll want shared into `__init__.py`
- When trying to import things from this directory, the file path you define can point at the directory instead of the individual files

Demo

1. Define 2 modules in a directory
   Make each one contain multiple variables defined on the local scope to be exported
2. Import everything into an `__init__.py` file that resides in the same directory as the 2 modules- thereby creating a package
3. Reference the package and use some of the exportable variables in the module

`main.py`

```
from helpers import api_key, shout_number  # absolute impor
print(api_key) # OUTPUT: 999
shout_number() # OUTPUT: 100
```

`helpers/__init__.py`

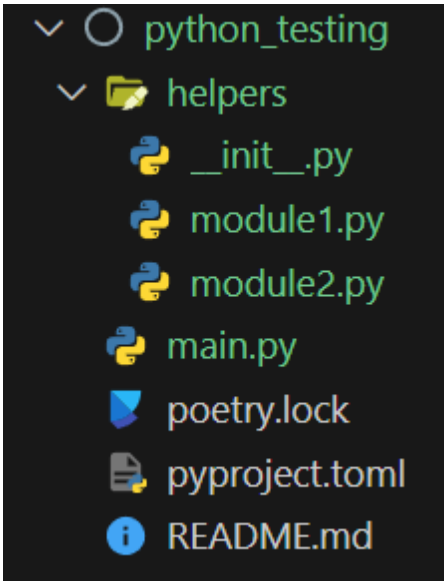
```
from .module1 import api_key         # relative import to sa
from .module2 import shout_number    # relative import to sa
```

`helpers/module1`

```
api_key = 999
```

`helpers/module2`

```
def shout_number():
    print(100)
```



Not needed after Python 3.3

After Python 3.3, you no longer need the init file to create packages

File Paths in Python vs JavaScript

File paths in python are specified using dot notation, unlike in JS

Differences

Dot Notation: Paths to Modules

This table could go on forever
Just be aware that you can reference modules in directories nested as high as you want, just by using the right # of leading dots

- Great grandparent directory syntax would involve 4 dots
  Great great grandparent directory syntax would involve 5 dots
- Number of directories we go up by = = = number of leading dots - 1

Dot Notation: Paths to Directories

This table could go on forever
Just be aware that you can reference directories nested as high as you want, just by using the right # of leading dots

- Great grandparent directory syntax would involve 4 dots
- Great great grandparent directory syntax would involve 5 dots

Absolute Imports

Absolute imports allow you to specify file locations using the project's root directory as the starting point
The root directory is the directory where the central python file is located - main.py in our examples

main.py

```
from helpers import api_key, shout_number  # absolute impor
print(api_key) # OUTPUT: 999
shout_number() # OUTPUT: 100
```

helpers/__init__.py

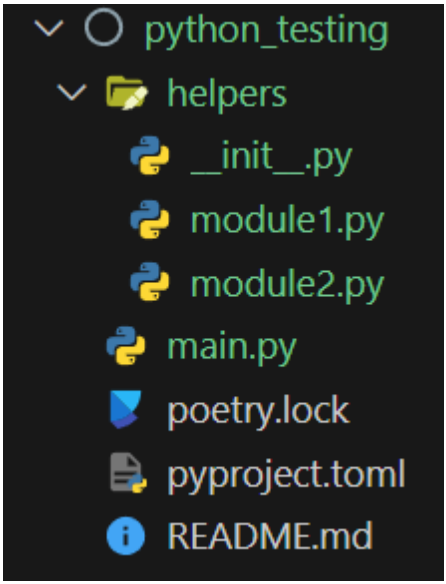
```
from .module1 import api_key         # relative import to sa
from .module2 import shout_number    # relative import to sa
```

helpers/module1

```
api_key = 999
```

helpers/module2

```
def shout_number():
    print(100)
```



Various ways to Import Content from a Module

We currently know how we can import things using both absolute and relative imports.
However, you have the option to import the following... (using relative or absolute)

1. Packages
2. Modules within packages
3. Methods within modules within packages
   Also works on variables, for the record

Absolute: Import Package

main.py

```
import package1  # absolute import the entire package
package1.run()   # access a method from the package
```

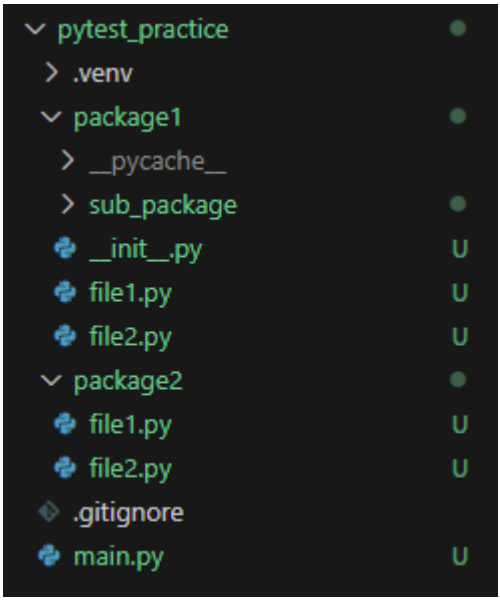
package1/file1.py

```
def run():
```

```
  print(88)
```

init.py (creates package)

```
from .file1 import run
```



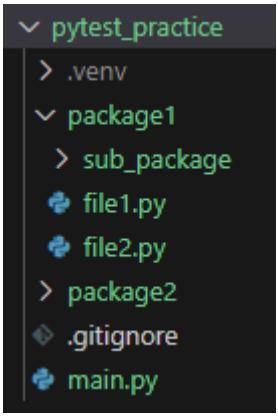
Absolute: Import Module

main.py

```
from package1 import file1  # absolute import the entire mo
file1.run()      # execute the function defined in package1/
```

package1/file1.py

```
def run():
  print(88)
```



Absolute: Import Method

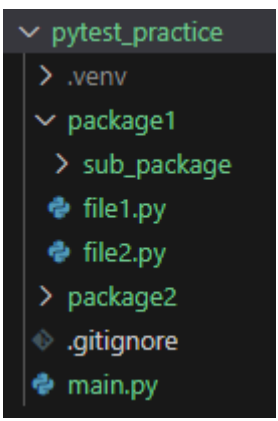
Import a global from the module, which is inside a package

main.py

```
from package1.file1 import run  # absolute import the metho
run()       # execute the function defined in package1/file
```

package1/file1.py

```
def run():
  print(88)
```



Relative Imports

Intro to Relative Imports

Import Module in Same Directory

main.py

```
from helpers import shout_api_key  # absolute import to hel
shout_api_key()  # OUTPUT: 90210
```

helpers/module1.py

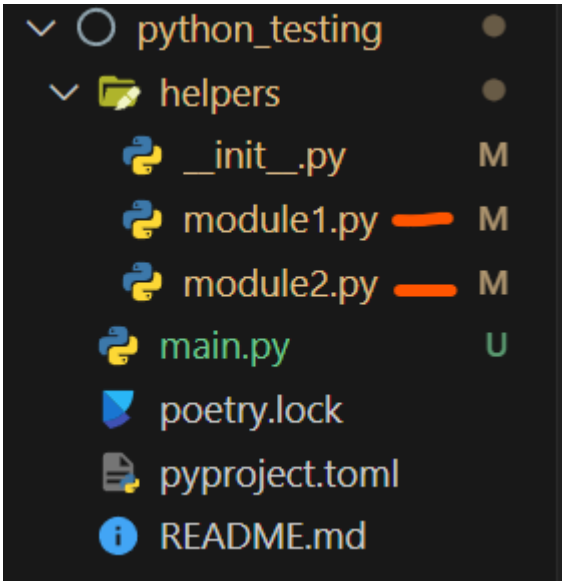
```
api_key = 90210
```

helpers/module2.py

```
from .module1 import api_key  # relative import module in s
def shout_api_key():
    print(api_key)
```

__init__.py

```
from .module1 import api_key          # relative import to m
from .module2 import shout_api_key    # relative import to m
```



Import Module in Child Directory

main.py

```
from helpers import shout_api_key  # absolute import to hel
shout_api_key()  # OUTPUT: 90210
```

helpers/subdir/module1.py

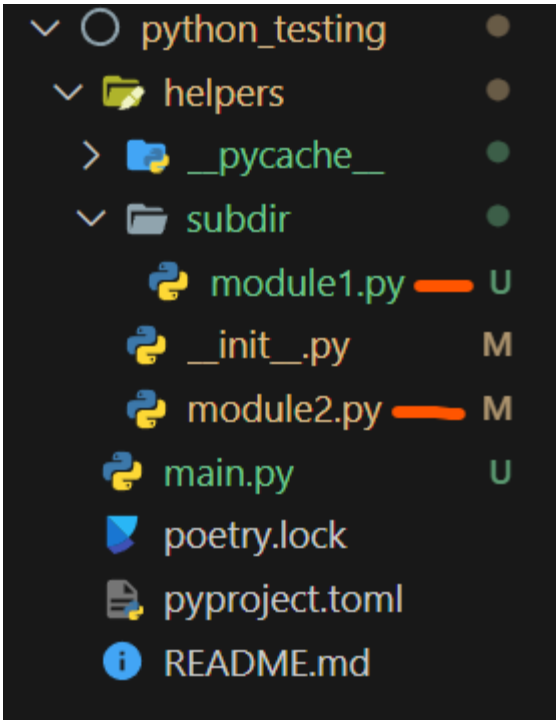
```
api_key = 90210
```

helpers/module2.py

```
from .subdir.module1 import api_key  # relative import modu
def shout_api_key():
    print(api_key)
```

helpers/__init__.py

```
from .subdir.module1 import api_key          # relative impo
from .module2 import shout_api_key      # relative import to
```



Import Module in Parent Directory

main.py

```
from helpers import shout_api_key  # absolute import to hel
shout_api_key()  # OUTPUT: 90210
```

helpers/subdir/module1.py

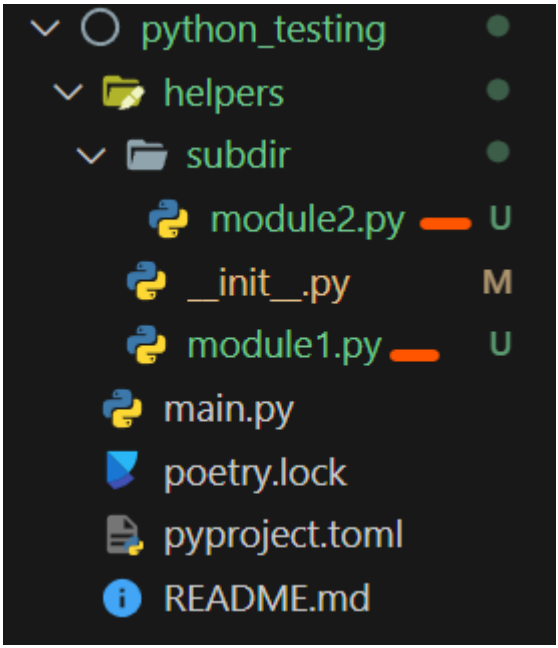
```
api_key = 90210
```

helpers/module2.py

```
from ..module1 import api_key  # relative import to module
def shout_api_key():
    print(api_key)
```

helpers/__init__.py

```
from .module1 import api_key                    # relative impo
from .subdir.module2 import shout_api_key       # relative impo
```



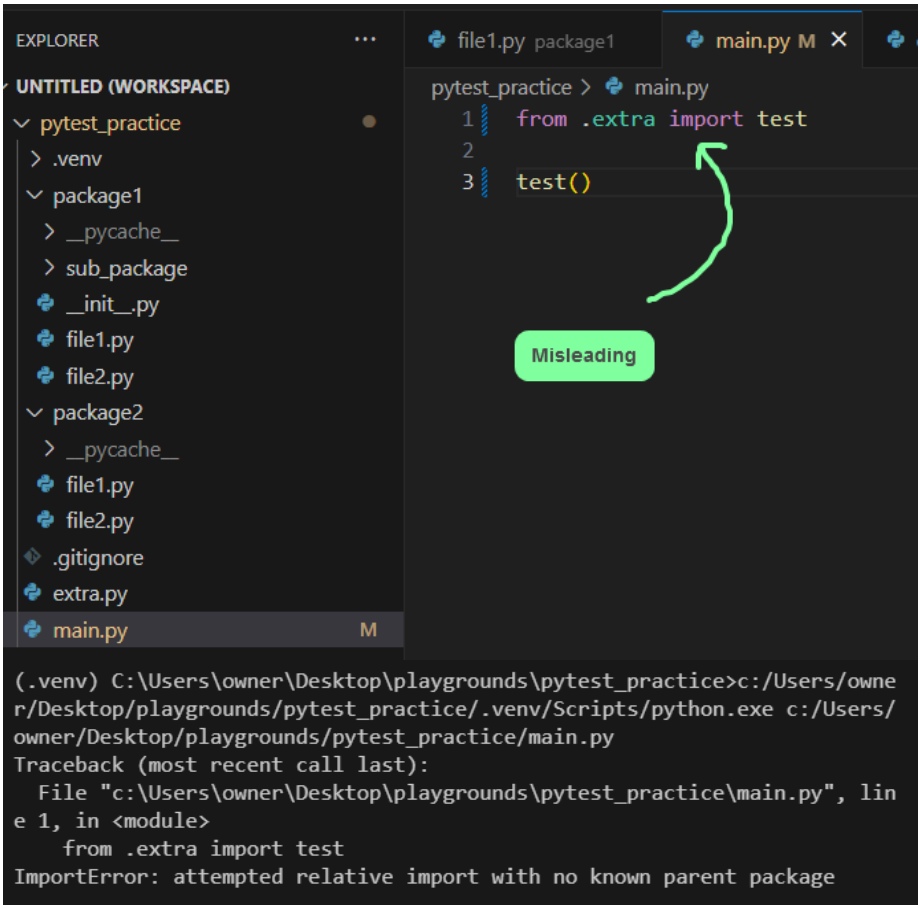
Why Python Relative Imports are so Strange (VSCode lies to you)

Relative imports can be confusing as Python thanks to the following reasons:

- Top level package rule is difficult to understand at first
  (this is limitation #1, which we focus on in the next lesson)

- VS Code oftentimes won't render any red error squiggly lines, even when a relative import is bound to fail
  This gives people a false sense of security until a script is ran which results in errors (IDE lies to you basically)

  EXAMPLE:
  Using relative imports on the root python file will not work (this is limitation #2, which we learn about later)
  Yet VS Code sees no problem with it until we run the script



Limitation #1: Top Level Package Rule

You cannot use relative imports to grab things from outside the package your python file currently resides in
Use absolute imports instead, or else you'll get an error saying this:

```
ImportError: attempted relative import beyond top-level package
```

Demonstration of Limitation

CAUSE OF ERROR

- In module1.py, we attempt to use a relative import to grab something from package2
- module1 is in package1, while module2 is in package2 (outside of

package1, where the relative import is written)

package1/module1.py

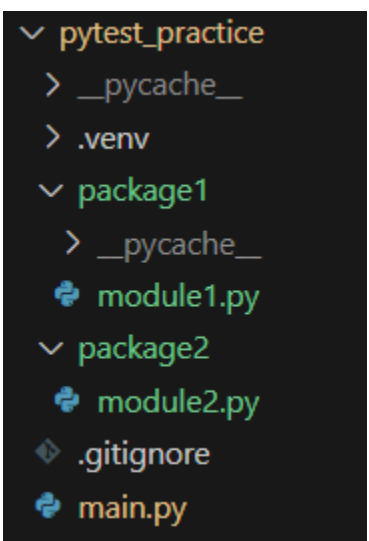
```python
from ..package2.module1 import pear # relative import (CAUS
def run():
    pear()
```

package2/module2.py

```python
def pear():
  print("I enjoy eating pears")
```

main.py

```python
from package1.module1 import run
run()
```




```
(.venv) C:\Users\owner\Desktop\playgrounds\pytest_practice>c:/Users/owner/Desktop/playgrounds/pytest_p
ractice/.venv/Scripts/python.exe c:/Users/owner/Desktop/playgrounds/pytest_practice/main.py
Traceback (most recent call last):
  File "c:\Users\owner\Desktop\playgrounds\pytest_practice\main.py", line 1, in <module>
    from package1.module1 import run
  File "c:\Users\owner\Desktop\playgrounds\pytest_practice\package1\module1.py", line 1, in <module>
    from ..package2.module1 import pear
ImportError: attempted relative import beyond top-level package

(.venv) C:\Users\owner\Desktop\playgrounds\pytest_practice>
```

Solution: Absolute Imports

Use absolute imports instead when you run into the top-level package error

package1/module1.py

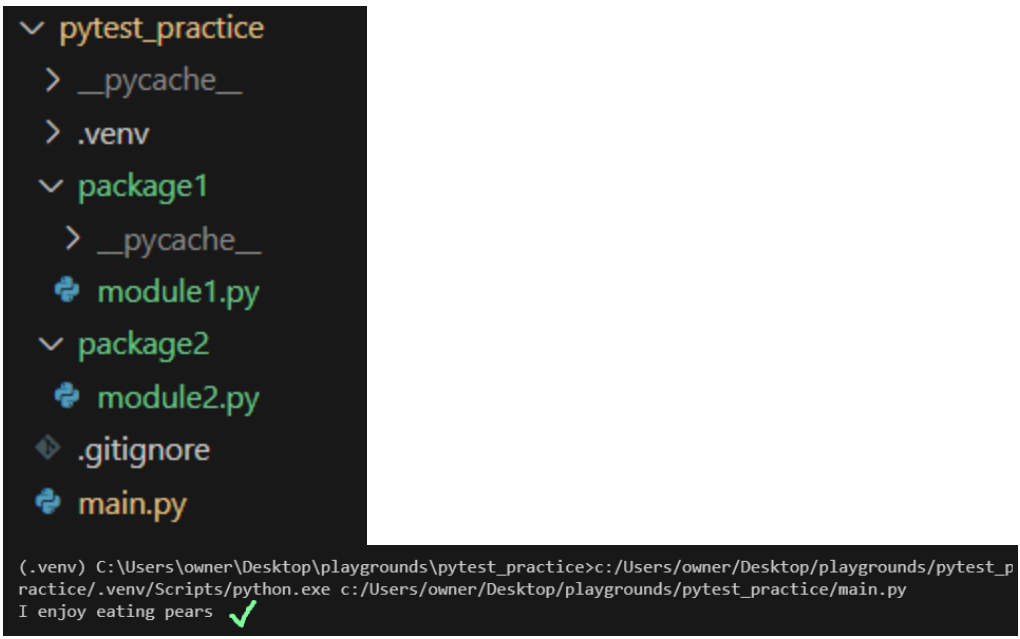
```
from package2.module1 import pear # absolute import (WORKS
def run():
    pear()
```

package2/module2.py

```
def pear():
  print("I enjoy eating pears")
```

main.py

```
from package1.module1 import run
run()
```



When you can Import from Higher Directories

Due to the top level package rule, it may seem like you can only use relative imports to target things from sibling or child directories
You actually can target files from parent directories, as long as they reside in the same package as the file you're writing in

EXAMPLE

- In module2, we use a relative import to target a method located in a module1
- module1 is in a parent directory of subpackage
- This works because both module1 and module2 are within the same package (package1)

package1/subpackage/module2.py

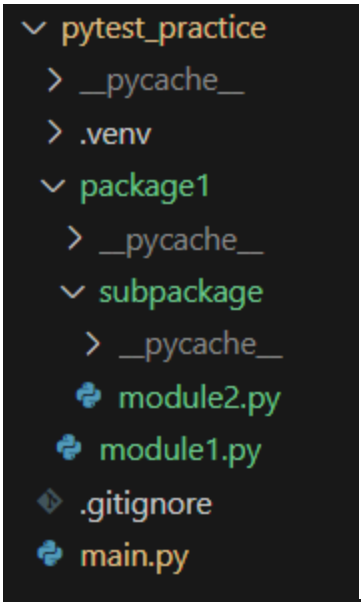
```
from ..module1 import pear # relative import method from a
def run():
    pear()
```

package1/module1.py

```
def pear():
  print("I enjoy eating pears")
```

main.py

```
from package1.subpackage.module2 import run  # absolute imp
run()
```




```
(.venv) C:\Users\owner\Desktop\playgrounds\pytest_practice>c:/Users/owner/Desktop/playgrounds/pytest_p
ractice/.venv/Scripts/python.exe c:/Users/owner/Desktop/playgrounds/pytest_practice/main.py
I enjoy eating pears
```

Limitation #2: Root Directory

You cannot use relative imports on the root directory of your Python projects

- Python figures that any top-level script should not have any parent packages that exist on a higher level
- Python won't assign it the package context required for relative imports to work
  This goes for relative imports to sibling, child, and parent directories (they will all fail)

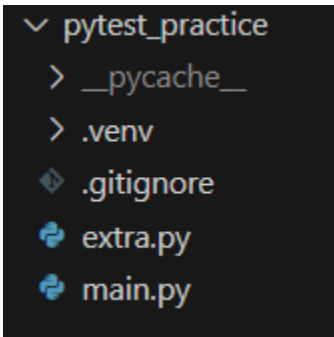
Demonstration

main.py

```
from .extra import test    # relative import the method from
test()
```

extra.py

```python
def test():
  print("Will I work?")
```




```
(.venv) C:\Users\owner\Desktop\playgrounds\pytest_practice>c:/Users/owner/Desktop/pla
ygrounds/pytest_practice/.venv/Scripts/python.exe c:/Users/owner/Desktop/playgrounds/
pytest_practice/main.py
Traceback (most recent call last):
  File "c:\Users\owner\Desktop\playgrounds\pytest_practice\main.py", line 1, in <modu
le>
    from .extra import test
ImportError: attempted relative import with no known parent package
```

Solution: Use Absolute Imports Instead

When on the root directory of your python project, you must use absolute imports

main.py

```python
from extra import test    # absolute import the method from
test()
```

extra.py

```python
def test():
  print("Will I work?")
```

Limitation #3: Circular Imports

Python will throw an error if 2 files import things from each other

- Module A imports from Module B
- Module B imports from Module A

CIRCULAR IMPORT ERROR MESSAGE

```
ImportError: cannot import name 'CreateModel' from partially
initialized module 'manager_server.routers.v1.create'
(most likely due to a circular import)
```

SOLUTION:
Make it so both files don't import things from each other
If both of them need something, create a separate file for them to both import things from

PIP: Package Manager for 3rd Party Modules

Intro to PIP

After Python 3.4, PIP comes installed automatically with Python - prior to that it was a separate tool

Check if PIP is Installed

Run command to see what version of PIP is installed (if any)

```
py -m pip --version
```

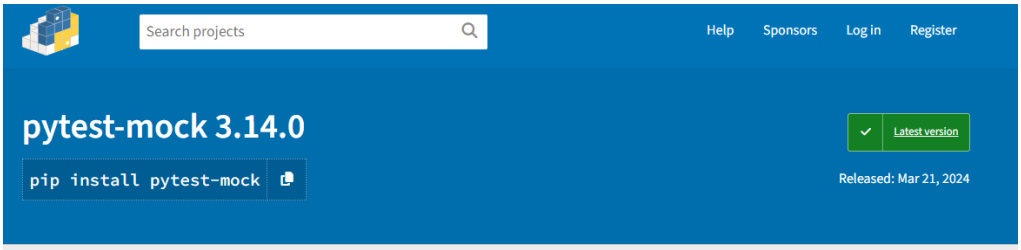
Where to find 3rd Party Packages

This is the Python ecosystem equivalent to NPM
https://pypi.org/

Pypi: Python Ecosystem's NPM

Finding Packages

The python ecosystem has a list of 3rd party packages available for use on a centralized platform, just like NPM
https://pypi.org/



Deciding what Packages are Safe

Base your opinions on how safe the package is to use on a few factors

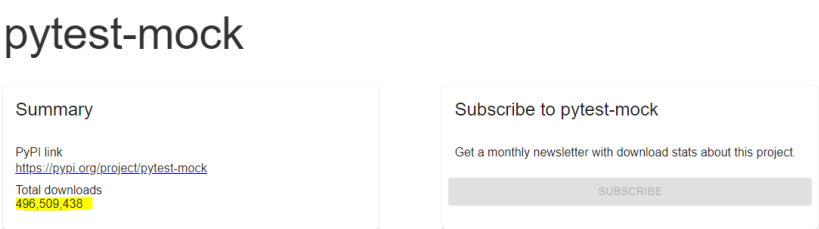
1. Popularity
   See how many downloads the package has

2. When the last update was released
   It should be somewhat recent

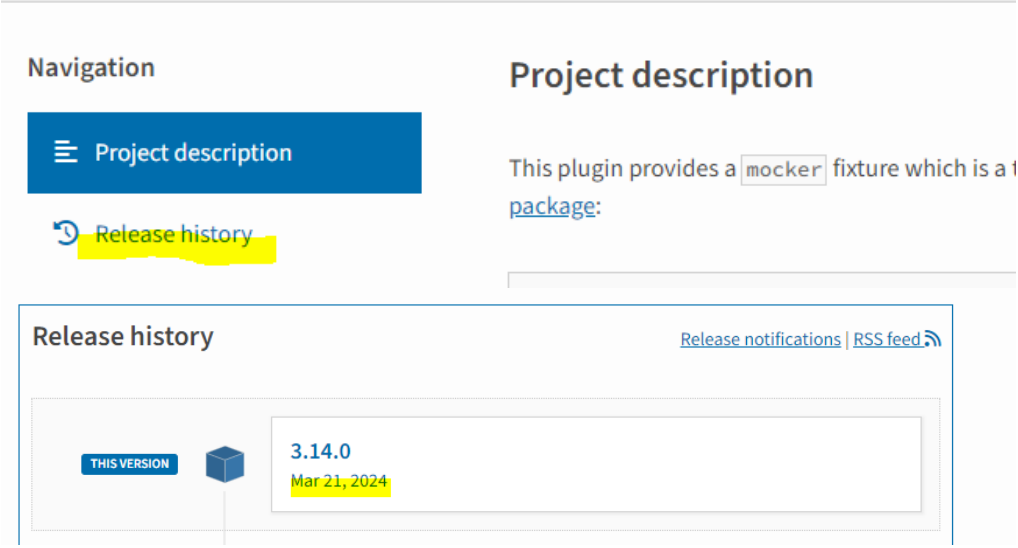
3. If the project is still being maintained - the dedicated docs page for a package usually has that info

Where to get the Info you Need

POPULARITY:
Pyi does not have exact download numbers on their package pages, but this external site does
https://www.pepy.tech/projects/pytest-mock



LAST UPDATE:
Pypi has a release history button



STILL BEING MAINTAINED:
The popular packages oftentimes have their own dedicated site for docs - usually mentoned in their PyPi descriptions (or easy to Google)
The site would tell you if a project is being abandoned
https://pytest-mock.readthedocs.io/en/latest/usage.html

Global Package Installs

You can take a dependency and install it globally on your machine
It will work on all projects, not just one (not always a good thing, which I'll explain in a later subsection)

Install Package Globally

Find a dependency you want on `pypi.org` then use the following syntax (only works if Python is added to PATH - setting that can be enabled during installation - uninstall the reinstall if this fails)

Install most recent version of package

```
py -m pip install python-dateutil
```

Install Specific Version of Package

```
py -m pip install python-dateutil==2.8.1
```

Update Package to whatever the newest version is

```
py -m pip install -U python-dateutil
```

Get List of Global Installs

Use either of these commands to get a list of global installs

```
pip list
pip freeze
```

Delete 1 Globally Installed Package

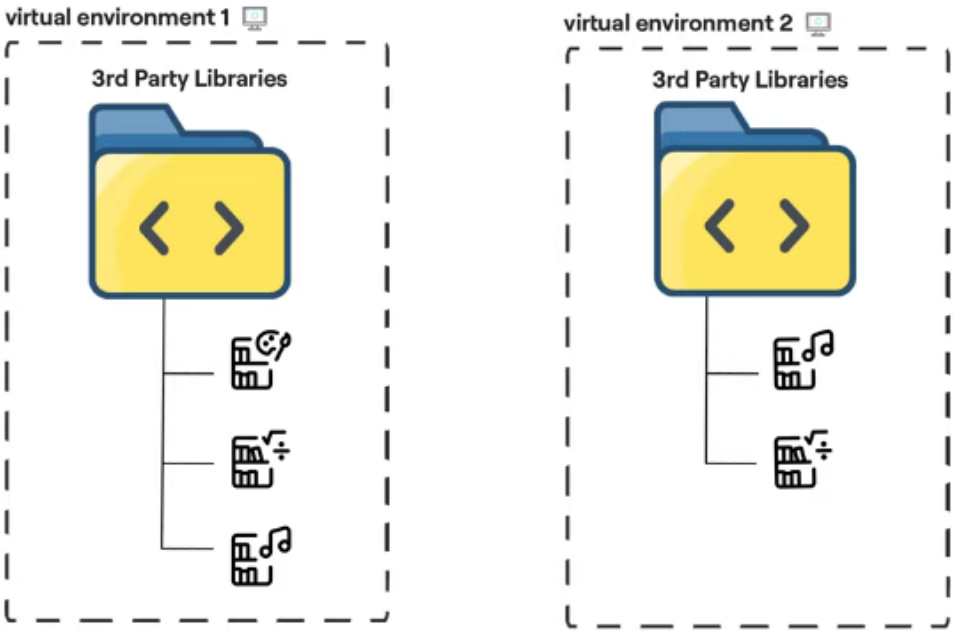
Look up what global installs you have first (see previous subsection)
Then use the following command to get rid of one

```
pip uninstall <package_name>
```

Warning: Don't Rely Global Installs for Professional Projects

When writing project backends, we should not rely on global package installs

- Project A might rely on V3.1.1 of a dependency, while Project B relies on V2.19.0
- Other developers will not be able to run the project since they probably don't have the same global package installations as you



SOLUTION:
Virtual environments (will be explained in a future lesson or chapter)

Managing Dependencies (MAIN)

Install pkg on Virtual Env

This installs a dependency on your virtual machine, not globally

1. Activate virtual env
2. Enter the following command

```
pip install requests  # installs requests library
```

Uninstall Pkg

1. Activate virtual env
2. Enter the following command

```
pip uninstall requests
```

WARNING:
Keep in mind installing 1 package will install other packages that are used to make it work
Check out other packages that you may not want using `pip list`

Poetry: Package Manager for 3rd Party Modules

Install Poetry w/ Windows Powershell

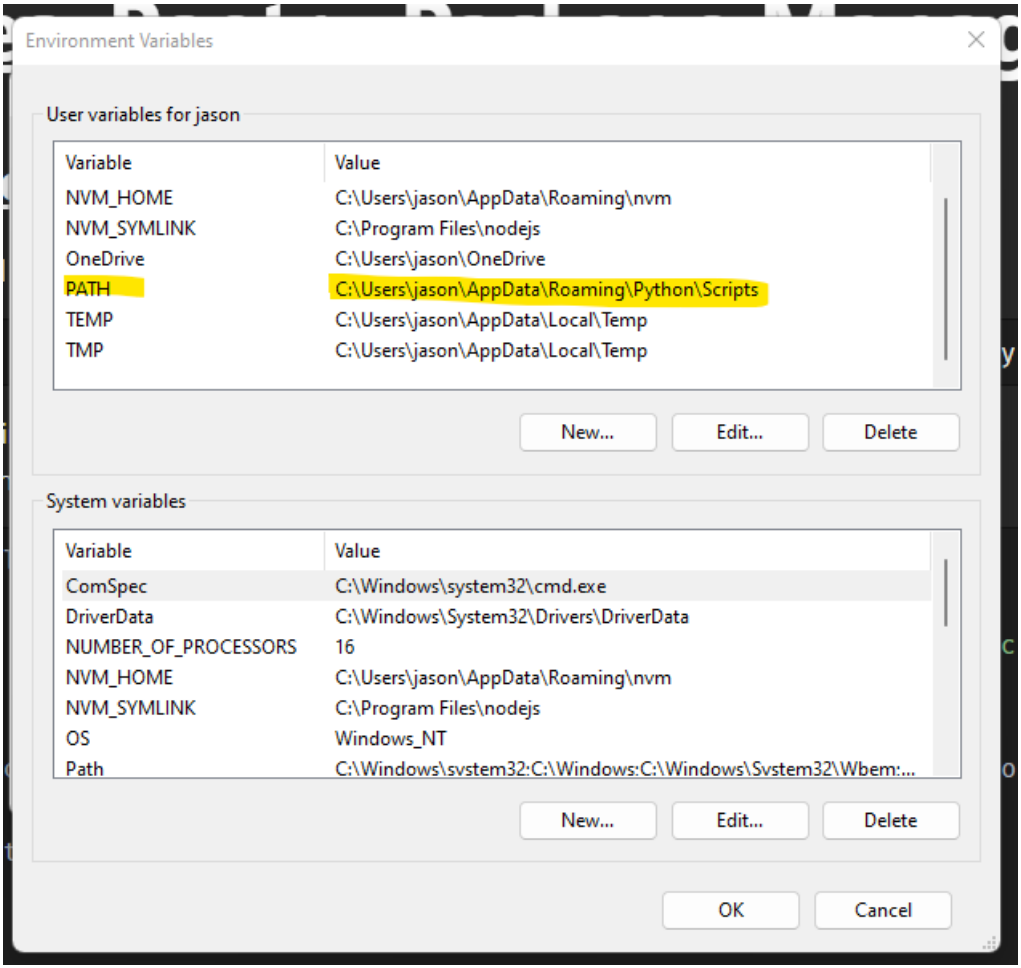
1. Open windows powershell and run this command

```
(Invoke-WebRequest -Uri https://install.python-poetry.org -
```

1. Look at the terminal and find the following success message
   Copy paste the path - in this case it's `C:\Users\jason\AppData\Roaming\Python\Scripts`

```
Poetry (1.7.1) is installed now. Great!

To get started you need Poetry's bin directory (C:\Users\jason
\AppData\Roaming\Python\Scripts) in your `PATH`
environment variable.

Alternatively, you can call Poetry explicitly with `C:\Users\jason
\AppData\Roaming\Python\Scripts\poetry`.

You can test that everything is set up by executing:

`poetry --version`
```

1. Create a new windows env variable named `PATH`, then set it equal to the path you copied from step 2
   https://www.youtube.com/watch?v=Gp_evQMHNDo



1. Open Windows Powershell again and run the following command

```
poetry --version
```

1. Go to your VS Code editor, open settings.json
    - Add the following, save, close VS Code, reopen it
    - Note that this is the other path provided in step 2's success message - with `\` replaced with `/`

```
"terminal.integrated.env.osx": {
  "PATH": "C:/Users/jason/AppData/Roaming/Python/Scripts/
}
```

1. Run the following in a VS Code terminal - if it works, then you're done

```
poetry --version
```

Use Poetry to Manage a Project

Part 1: Initialize an existing repo

1. Use your VS Code terminal to navigate into your existing repo, then run `poetry init`

```
cd pre-existing-project
poetry init
```

1. Answer all the questions it asks - similar to `npm init`

FASTER SETUP (skips all questions)

```
cd pre-existing-project
poetry init -n
```

Part 2: Add Dependencies

1. Find a dependency on https://pypi.org/
   The install command will usually look like the following

```
pip install python-dateutil
```

1. Replace `pip install` with `poetry add`, then run it in your project's central directory

```
poetry add python-dateutil
```

1. Once all your poetry dependencies have been added, run the following command
   It installs all dependencies in `pyproject.toml` (equivalent to npm install)

```
poetry install
```

Part 3: Add Dev Dependencies

Some dependencies are required by the dev team, but not the project in production mode

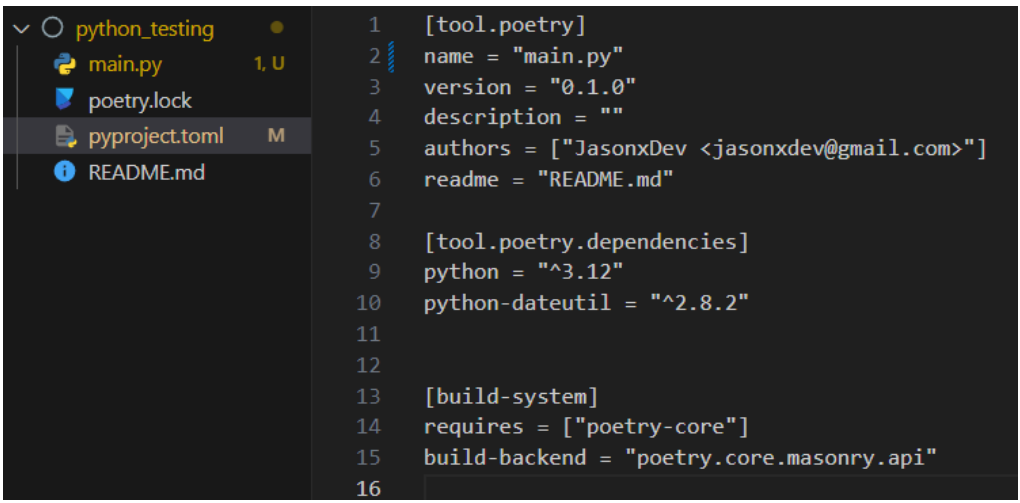
Ex. Packages that help enforce type checking

```
poetry add --group dev mypy
```

Part 4: Run your Scripts

AFTER MATH
By now you should have 2 new files: `poetry.lock` and `pyproject.toml`

- The `toml` file should contain your dependencies and answers to the config questions you answered when initializing the project
- When you add or remove dependencies, they will appear here



TEST YOUR SCRIPT BY RUNNING IT
To run a script that relies on dependencies, use the following command
Change the script path depending on your script's file name and the directory where you're running this command

```
poetry run python main.py
```

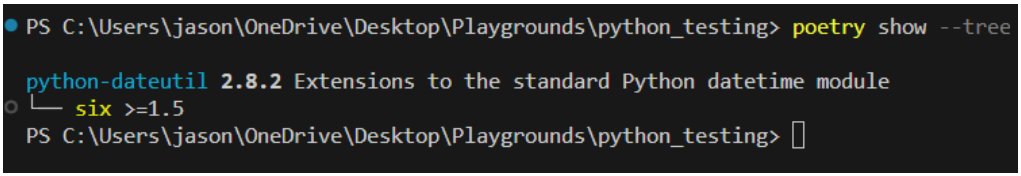
ACTIVATE POETRY SHELL:
https://youtu.be/Ib7fNOIGM7E?t=476

- Technically you can stop here, but the command above gets tedious to run or paste every time
- You can run a poetry shell then run scripts easier in the near future (just keep this terminal open)

```
poetry shell
python main.py
```

Show Installed Python Packages

```
poetry show     (flat)
poetry show --tree   (nested structure)
```


PS C:\Users\jason\OneDrive\Desktop\Playgrounds\python_testing> poetry show --tree
python-dateutil 2.8.2 Extensions to the standard Python datetime module
six >=1.5
PS C:\Users\jason\OneDrive\Desktop\Playgrounds\python_testing>

Remove Python Packages

Remove dateutil dependency

```
poetry remove python-dateutil
```

Sources

Poetry Guide
https://www.youtube.com/watch?v=Ib7fNOIGM7E

Virtual Environments

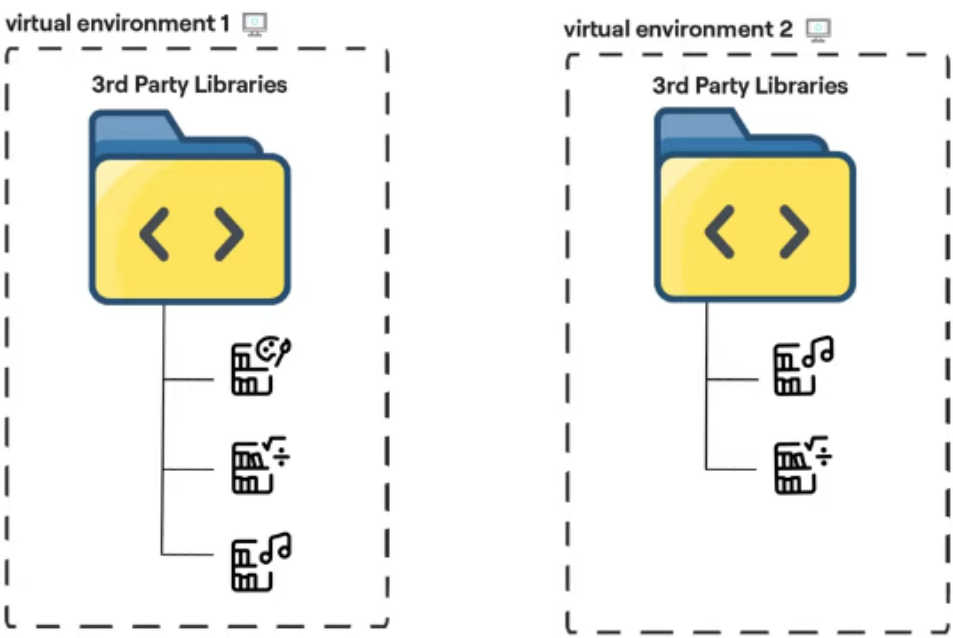
Why Virtual Envs are Required

Most professional Python projects use dependencies
We install 3rd party packages so we can use other people's code without rewriting it ourselves

Why we need Virtual Environments

When writing project backends, we should not rely on global package installs

- Project A might rely on V3.1.1 of a dependency, while Project B relies on V2.19.0
- Other developers will not be able to run the project since they probably don't have the same global package installations as you



How Python Handles Dependencies Compared to JS

JAVASCRIPT
We have the `node-modules` folder + package.json to share our dependencies and versions with other devs

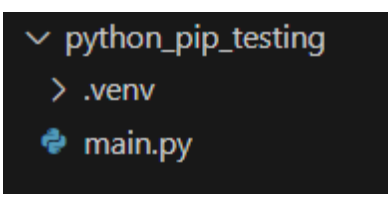
PYTHON
Virtual environments serve the exact same purpose, but their solution to the problem differs a bit
We do not send virtual environments to the repository on Github, just like node-modules

Virtual Env Usage Guide

Setup: Create .venv

Open the terminal in your project's root directory and create a `.venv` folder there

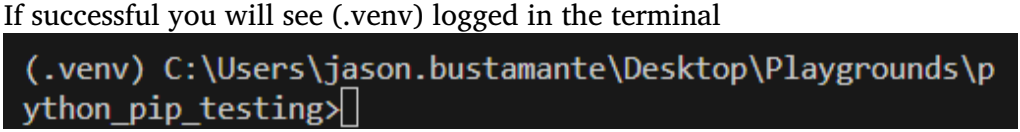
```
py -m venv .venv
or
python -m venv .venv
```



Activate Virtual Env

```
source .venv/Scripts/activate   (for Linux users)
.venv\Scripts\activate     (for Windows users)
source .venv/bin/activate    (WSL)
```

If successful you will see (.venv) logged in the terminal



Deactivate Virtual Env

To deactivate the virtual environment, use the command below

This will return your terminal to the global env

```
deactivate
```

List Dependencies in Virtual Env

1. First, activate your virtual environment (see previous subsection)
2. Then run 1 of these package listing commands

```
pip list
pip freeze
```

Manage Dependencies in Virtual Environments (PIP)

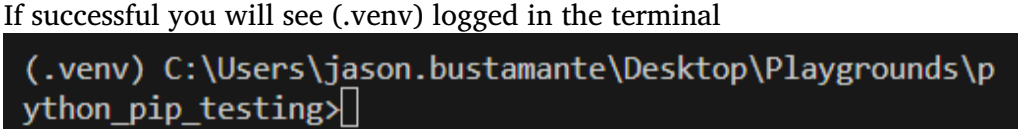
ACTIVATE VIRTUAL ENV FIRST

Before you use any of these commands, do the following
If these conditions are not met, you will perform actions on the global environment

1. Have a virtual environment set up (.venv folder should be on project root)
   If this is not done, see the previous lesson
2. Activate your virtual environment first

```
source .venv/Scripts/activate   (for Linux users)
.venv\Scripts\activate     (for Windows users)
```

If successful you will see (.venv) logged in the terminal



List Dependencies

```
pip list
pip freeze
```

Install Packages

Install most recent version of package

```
py -m pip install python-dateutil
```

Install Specific Version of Package

```
py -m pip install python-dateutil==2.8.1
```

Update Package to whatever the newest version is

```
py -m pip install -U python-dateutil
```

Uninstall Specific Package

```
pip uninstall <package_name>
```

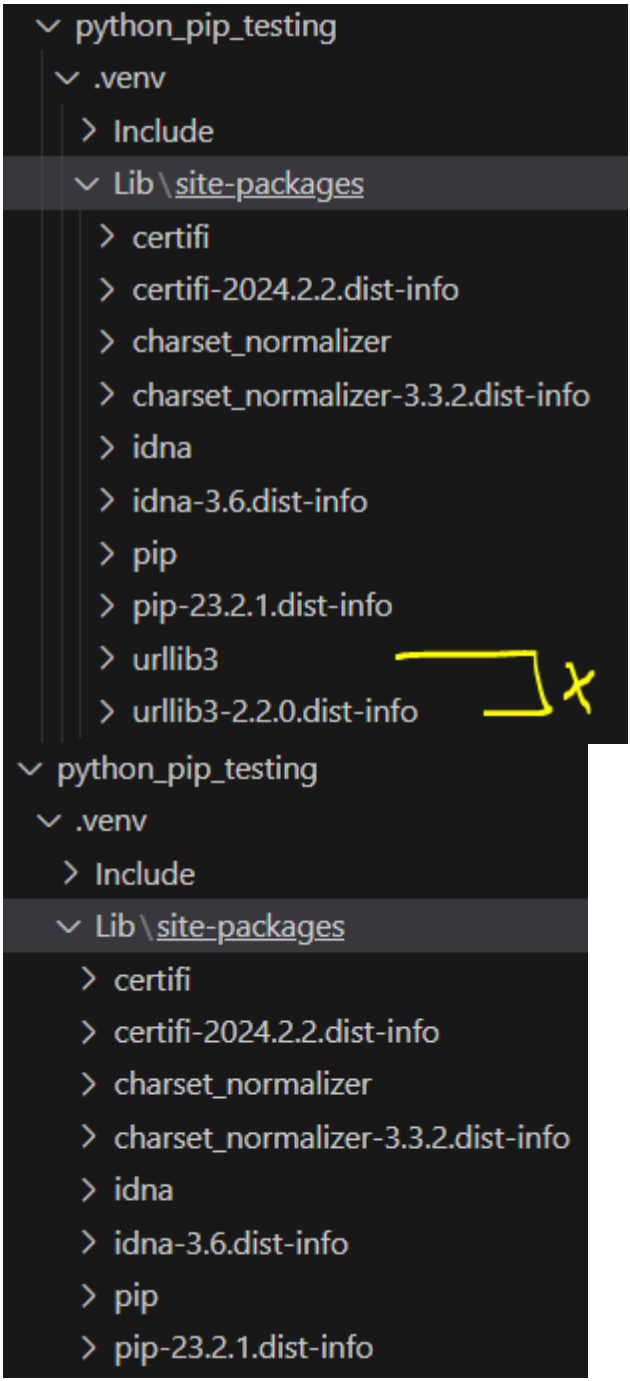
Visual Feedback when Managing Packages

As you delete packages, you can look inside `.venv/Lib/site-packages`

- Adding dependencies will add to this list of folders
- Uninstalling dependencies will take away from this list of folders

Example: Uninstall `urllib3`

```
# Virtual env is activated already
pip uninstall urllib3
```



Manage Dependencies in Virtual Environments (Poetry)

ACTIVATE VIRTUAL ENV FIRST

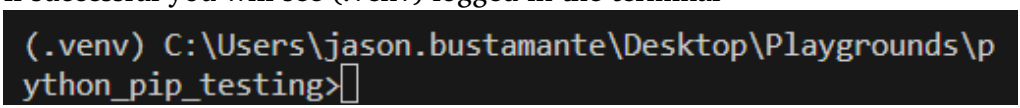
Before you use any of these commands, do the following
If these conditions are not met, you will perform actions on the global environment

1. Have a virtual environment set up (.venv folder should be on project root)
   If this is not done, see the previous lesson

2. Activate your virtual environment first

```
source .venv/Scripts/activate    (for Linux users)
.venv\Scripts\activate      (for Windows users)
```

If successful you will see (.venv) logged in the terminal



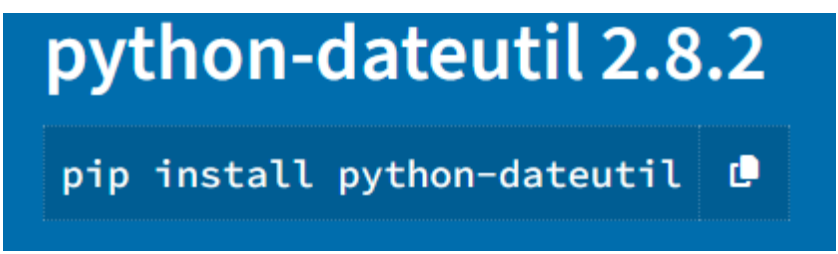
List Dependencies

```
poetry show     (flat)
poetry show --tree  (nested structure)
```

Install Packages

Install most recent version of package

```
poetry add python-dateutil
```



Install Specific Version of Package

```
poetry add django==2.0.5
```

Install all packages specified in `pyproject.toml` (equivalent to npm install)

```
poetry install
```

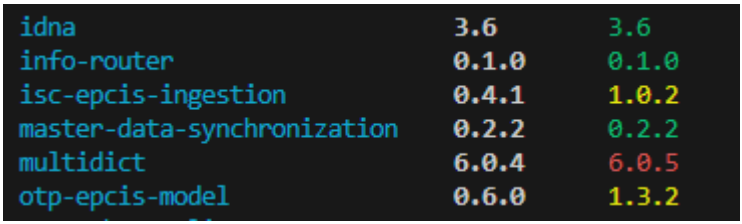
Uninstall Package

```
poetry remove python-dateutil
```

Update Package to Newer Version

1. Use this command to show which packages are outdated
   It takes a while if you have many dependencies

```
poetry show -l
```



1. Install a dependency using the specific version number on the right (most recent)

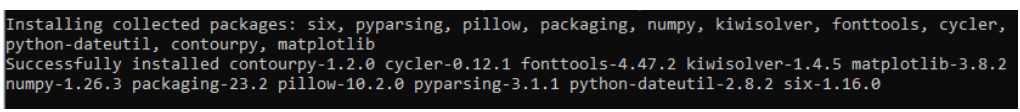
```
poetry add multidict==6.0.5
```

1 Package Contains Multiple Dependencies

If you run a single install command, it's likely you're installing more than just 1 package

1 Install, Multiple Dependencies

Installing a package globally means you also globally install whatever dependencies that package has
In the image below, we installed Matplotlib, which resulted in us globally installing a bunch of other stuff


Installing collected packages: six, pyparsing, pillow, packaging, numpy, kiwisolver, fonttools, cycler, python-dateutil, contourpy, matplotlib
Successfully installed contourpy-1.2.0 cycler-0.12.1 fonttools-4.47.2 kiwisolver-1.4.5 matplotlib-3.8.2 numpy-1.26.3 packaging-23.2 pillow-10.2.0 pyparsing-3.1.1 python-dateutil-2.8.2 six-1.16.0

Deleting 1 Dependency will take Multiple Commands

If you wish to uninstall matplotlib, it'll likely take 6 commands

You'll need to delete every dependency of the library that was installed

Managing Dependency Conflicts

What is a Dependency Conflict?

In a python project, we install 3rd party packages that have a version number

- Packages can have their own dependencies (sub dependency)
  And those dependencies can have their own as well.. etc etc
- Sometimes dependencies or sub dependencies can have conflicting versions- meaning 2 or more may require a package with different version numbers

Guide to Resolving Conflicts

Find which dependency or sub dependency requires a problematic version, then upgrade or downgrade it to bring its requirements in line with all the others
https://docs.divio.com/how-to/debug-dependency-conflict/

Resources for Resolving Conflicts

1. In `poetry.lock`, you can easily tell what packages and what subpackages there are (the file is structured like this)

```
[[package]]
[package.dependencies]
[package.extras]
[package.source]
```

1. You can see what subpackages a package has by visiting this website: https://libraries.io/pypi/httpx/0.28.1
   - Be aware that as the package version changes, so do its dependencies
   - Example:
     `httpx 0.23.3` requires `httpcore 0.16.3`. But `httpx 0.28.1` requires `httpcore 1.*`

httpx

Release 0.28.1

The next generation HTTP client.

Homepage - PyPI - Python

Dependencies

anyio	*
certifi	*
httpcore	==1.*
idna	*

Explore dependencies

= = = = = = = = ADV BASICS = = = = = = = =

Dates

Python does not have a built in data type for dates
We typically import the datetime package which is built into Python

```
import datetime
```

Date vs Datetime Objects + Extracting their Details

Python has a built in package called datetime, and they have 2 main classes: `date` and `datetime`
These classes generate either date or datetime objects

date Objects

Represents a date (year, month, and day) without any time information.

- Instances of the `date` class have attributes like `year`, `month`, and `day` to access individual components of the date.
- Useful for tasks that only require working with dates, such as handling birthdays, anniversaries, or historical events.

```
from datetime import date
today = date.today()
print(today)      # Output: 2023-12-27
print(today.year)  # Output: 2023
print(today.month) # Output: 12
print(today.day)   # Output: 27
```

datetime Objects

Represents both a date and a time (year, month, day, hour, minute, second, microsecond) and is more precise.

- Instances of the `datetime` class can store information about both dates and times, making it suitable for tasks that require precise timestamp information.
- Provides attributes like `year`, `month`, `day`, `hour`, `minute`, `second`, and `microsecond` for accessing individual components of both date and time.

```
from datetime import datetime
now = datetime.now()
print(now)              # Output: 2023-12-27 12:34:56.789012
print(now.year)         # Output: 2023
print(now.month)        # Output: 12
print(now.day)          # Output: 27
print(now.hour)         # Output: 12
print(now.minute)       # Output: 34
print(now.second)       # Output: 56
print(now.microsecond)  # Output: 789012
```

Create Date Objects

Current Date

Get current date in YYYY-MM-DD format

```
from datetime import date
# Declare date object
```

```
today = date.today() # date object for today
# Convert to string
date_string = today.strftime("%Y-%m-%d") # 2023-12-25
```

Specific Dates

WARNING:
Only use this when time zone differences do not matter
If they do matter, check the "Create datetime Objects" lesson

```
from datetime import date
# Declare a date object
x = date(1996, 8, 10)            # date object for 1996-08-1
# Convert the date object into an ISO string
y = date(1996, 8, 10).strftime("%Y-%m-%dT%H:%M:%SZ") # 1996
```

Keep in mind that variable `y` holds a date at exactly midnight

- Usually when we deal with dates that someone selects, we need to take their time zones into account and impose an hourly offset
- A datetime would likely be more practical for that

Create datetime Objects

Current Timestamp

Get your current timestamp, and convert it to UTC time
(the hourly offset will be applied)

```
from datetime import datetime, timezone # timezone changes
now = datetime.now(timezone.utc).strftime("%Y-%m-%dT%H:%M:%
print(now)   # 2023-12-27T21:28:24Z
```

HOURLY OFFSET EXPLANATION:

At the time of writing this, it was December 27th, 2023. At 4:28PM
The now variable was set to T21 because EST is 5 hours behind UTC time at this time of year
So in UTC time it was actually 9PM, represented by T21 since 12 + 9 = 21

Specific Timestamp (frontend only)

Use this method to create date strings in ISO format (`2023-12-25T05:00:00.000Z`)

- It takes the user's local timezone into consideration
- The hourly offset will be applied, and the date string will represent the true UTC time

```
from datetime import datetime, timezone
def create_utc_datetime(year, month, day, hour=0, minute=0,
    # Create a datetime object with the provided parameters
    dt = datetime(year, month, day, hour, minute, second, m
    # Assuming the provided datetime is in local time, conv
    dt_utc = dt.astimezone(timezone.utc) # Note: This assum
    # Format the datetime in the specified format
    formatted_utc_datetime = dt_utc.strftime("%Y-%m-%dT%H:%
    return formatted_utc_datetime
```

Example: Dec 25th, 2023 (when Toronto is 5 hrs behind UTC time)

```
create_utc_datetime(2023, 12, 25)  # 2023-12-25T05:00:00.00
```

Example 2: Aug 7th, 2023 at noon (when Toronto is 4 hrs behind UTC time)

```
create_utc_datetime(2023, 8, 7, 12)  # 2023-08-07T16:00:00.
```

Why Specific Timestamp Snippet Fails on Backend

The `timezone` import will be different depending on where your script is run

WHY THIS FAILS ON BACKEND

- You can write a Python script that uses `timezone`, and submit it to your org's repository
- Your code will generate a different result when you run code on your local computer vs when the same code runs on a sever in Brazil

Why Specific Timestamp Snippet Works on Frontend

HOW TO HANDLE DATES ON THE FRONTEND
The `timezone` working differently is a behavior we actually want on the frontend

- If a user selects a timestamp, we want it to be relative to their time zone - which is likely a few hours off UTC time
- Send the backend a string in ISO format with the hourly offset applied already

EXAMPLE:
User located in Canada selects a timestamp of Dec 25th at 1AM on the frontend

- User has selected Dec 25th at 1AM in their time zone (EST), which is 5 hours behind UTC at that time of the year
- Should send the backend the following date string: `2023-12-25T06:00:00.000Z`

Format Codes: Use to Build Formatted Date Strings

Directives table

The following directives can be used in combination with the `strftime()` method - which can be used on date or datetime objects
You can create your own formatted date strings with these

Create Strings using Directives

Use strftime with the directives above to create your own formatted date

strings

Take current date
Make a date string with MM/DD/YYYY format

```
from datetime import date
today = date.today().strftime("%m/%d/%Y") # notice we add /
print(today)  # 12/25/2023
```

Take Current date
Make a date string with Weekday, Month, Day, Year spelt in English

```
from datetime import date
today = date.today().strftime("%A %B %d, %Y")
print(today)  # Monday December 25, 2023
```

Works on any Date or Datetime Objects

In our examples we stuck to using date objects for the current day

- In practical applications we'll likely receive date strings
- You should convert them to date or datetime objects first, then format with strftime()
  (this is the focus of the next lesson)

Turn Date Strings into Date or Datetime Objects

BEFORE YOU BEGIN
Decide whether you want a date or datetime object

2020-01-21

Turn into date object

```
from datetime import date
```

```
date_string = "2020-01-21" # date string in YYYY-MM-DD form
date_obj = date.fromisoformat(date_string) # date object
```

Turn into datetime object

```
from datetime import datetime, timezone
date_string = "2020-08-21" # date in YYYY-MM-DD format
# Create date object
date_obj = datetime.fromisoformat(date_string) # date objec
# Create date object with local timezone's hourly offset ap
date_obj_utc = (datetime.fromisoformat(date_string)).astime
```

2023-12-25T22:40:08Z

Turn into date object

```
from datetime import date, datetime
date_string = "2023-12-25T22:40:08Z"
datetime_obj = datetime.fromisoformat(date_string)
date_obj = datetime_obj.date()
```

Turn into datetime object

```
from datetime import datetime, timezone, timedelta
date_string = "2023-12-25T22:40:08Z"
# date_obj and date_obj_utc are the same since the string w
date_obj = datetime.fromisoformat(date_string) # option 1
```

```
date_obj_utc = date_obj.astimezone(timezone.utc) # option 2
```

Edit Date Objects

We can take existing date objects and edit them to change their year, month, day, etc

Why we use dateutil Library

Dateutil library is more flexible than Python's built in datetime module

- Python's built in datetime module allows us to edit dates, but there are limitations
  Ex. A value error gets thrown if you try to add 2 months to a date in December, since that rolls over into the following year
- Dateutil is smart enough to handle commands like these

Demo

Take date object for today
Set the year to 4 years before

```
from datetime import date
from dateutil.relativedelta import relativedelta
# Take object for today's date and reverse it 4 years using
today = date.today()  # Date object for 2023-12-27 (today a
fourYearsAgo = today + relativedelta(years=-4)  # Date obje
# Now convert the date object to a string
dateString = fourYearsAgo.strftime("%Y-%m-%d")  # 2019-12-2
```

Take a date object for today
Set the month to 2 months later

```
from datetime import date
from dateutil.relativedelta import relativedelta
# Take object for today's date and fast forward 2 months us
today = date.today()  # Date object for 2023-12-27 (today a
twoMonthsLater = today + relativedelta(months=2)  # Date ob
# Now convert the date object to a string
dateString = twoMonthsLater.strftime("%Y-%m-%d")  # 2024-02
```

Take a date object for today
Set the day to 1 day before

```
from datetime import date
from dateutil.relativedelta import relativedelta
# Take object for today's date and fast forward 2 months us
today = date.today()  # Date object for 2023-12-27 (today a
yesterday = today + relativedelta(days=-1)  # Date object f
# Now convert the date object to a string
dateString = yesterday.strftime("%Y-%m-%d")  # 2023-12-26
```

Day Difference b/t Date Objects

You can get the day difference between 2 date objects

Demo

Get the day difference between 2 dates

```
from datetime import datetime
# Define two date objects
date1 = datetime(2023, 12, 15)
date2 = datetime(2020, 12, 27)
# Calculate the day difference
day_difference = (date2 - date1).days      # - 1083 (since d
absolute_day_difference = abs((date2 - date1).days) # 1083
```

It even works on date objects returned from methods in other date packages, like dateutil in this case

```
from datetime import date
from dateutil.relativedelta import relativedelta
# Take object for today's date and fast forward 2 months us
today = date.today()  # Date object for 2023-12-27
yesterday = today + relativedelta(days=-20)  # Date object
day_difference = abs((today - yesterday).days) # 20
```

Sorting Lists of Dates

Sorting Date Objects

Simply use the sorted() method

```
from datetime import datetime
```

```
# Define two date objects
date1 = datetime(2023, 12, 15)
date2 = datetime(2019, 9, 27)
date3 = datetime(2015, 6, 1)
# Create a list of date objects
dates_list = [date1, date3, date2]
# Sort the dates_list from oldest to most recent
sorted_dates_list = sorted(dates_list)
```

Sorting Date Strings in YYYY-MM-DD Format

```
from datetime import datetime
# Define a list of date strings
date_strings = ["2023-12-15", "2020-12-27", "2021-05-10", "
# Convert date strings to datetime objects
date_objects = [datetime.strptime(date_str, "%Y-%m-%d") for
# Sort the date objects from oldest to most recent
sorted_date_objects = sorted(date_objects)
# Convert sorted date objects back to date strings if neede
sorted_date_strings = [date.strftime("%Y-%m-%d") for date i
print(sorted_date_objects)
print(sorted_date_strings) # ['2019-09-01', '2020-12-27', '
```

Sorting Date Strings in ISO Format

```python
from datetime import datetime
# Define a list of date-time strings
date_strings = ["2023-12-15T05:40:08Z", "2019-12-27T11:20:2
# Convert date-time strings to datetime objects
date_objects = [datetime.strptime(date_str, "%Y-%m-%dT%H:%M
# Sort the date objects from oldest to most recent
sorted_date_objects = sorted(date_objects)
# Convert sorted date objects back to date-time strings if
sorted_date_strings = [date.strftime("%Y-%m-%dT%H:%M:%SZ")
print(sorted_date_strings) # [2019-12-27T11:20:22Z, 2021-05
```

Get Hourly Offsets for Different Locations

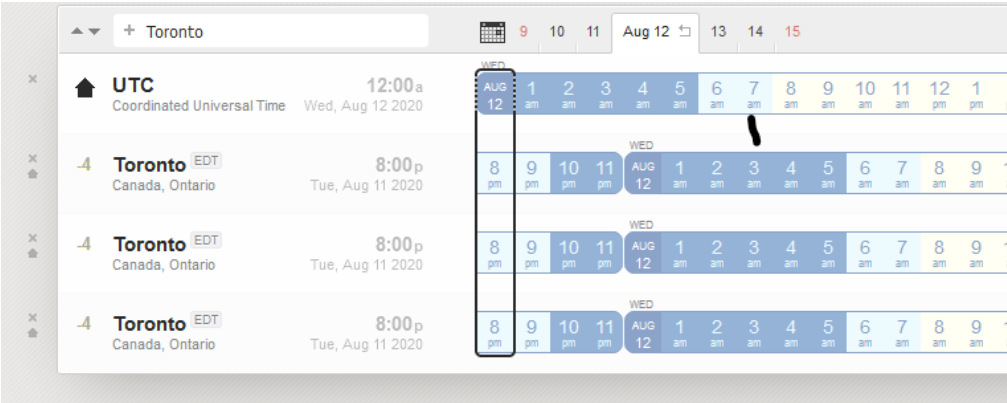
Site Link

https://www.worldtimebuddy.com/

Specify the city and the exact date
The site will tell you how many hours behind or ahead of UTC that location's time zone is

Example

We can see 7AM at UTC time is 3AM in Toronto on August 12th
Means Toronto is 4 hours behind UTC time (EST is -04:00)



Custom Util Methods

Current Timestamp: ISO

This method generates an ISO timestamp for this exact moment
This code runs on the server, which is likely in another time zone, but that's okay since the timestamp is in UTC

```
from datetime import datetime, date, timezone
def current_timestamp() -> str:
    return datetime.now(timezone.utc).strftime("%Y-%m-%dT%H
print(current_timestamp()) # 2024-09-13T21:33:51Z
```

Convert ISO Timestamp to Datetime Object

This takes an ISO timestamp string and converts it to a Python date object

- Keep in mind this method only works for ISO timestamps that factor up to seconds
  Ex. `2024-09-13T21:33:51Z`

- If your timestamp factors up to milliseconds you'll need to edit this method a bit first
  Ex. `2024-09-13T21:36:08:36Z`

```
from datetime import datetime, date, timezone
def iso_to_datetime_object(iso_timestamp):
    return datetime.strptime(iso_timestamp, "%Y-%m-%dT%H:%M
iso = "2024-09-13T21:30:03Z"
print(iso_to_datetime_object(iso))  # UTC time date object
```

Check if timestamp is in Range of 2 Others

This method works with datetime objects, not ISO strings

```
from datetime import datetime, timezone
def timestamp_is_between(
    target_date: datetime, start_date: datetime, end_date:
) -> bool:
    return start_date <= target_date < end_date
```

Times

Delay Code Execution for N seconds

You can pause the execution of code for N seconds

```
import time
print("Start")
time.sleep(1)  # Delays for 1 second
print("End")
```

Create Time Object

We declare time objects using a 24 hour format
Can convert this to a time string instead, using either 24 or 12 hour formats

```
from datetime import time
# Declare time object with parameters: # hours, minutes, se
time_object = time(15, 33, 8)  # Time object for 15:33:08
# Turn time object into time strings (24 hr format and 12 h
time_string_24hr = time_object.strftime("%H:%M:%S")  # 15:3
time_string_12hr = time_object.strftime("%I:%M:%S %p")  # 0
```

< Combine Date Object + Time Object >

Error Handling

Definitions

EXCEPTIONS
Errors we can handle with an except block (python equivalent of a catch block)

ERRORS:
Syntax errors screw over the code completely and stop it from running
No error handling will allow you to get past one of these

Raise Errors on Purpose

You can intentionally throw an error using the raise keyword followed by an error type

Value Error

```
raise ValueError # generic error
raise ValueError("Division by zero is not allowed.") # valu
```

Can raise other error types as well - with or without a message (covered in next lesson)

```
raise SyntaxError
raise NameError
raise IndexError
raise TypeError
raise KeyError
```

Common Error Types

Common Error Types

Each of these errors indicates a different kind of issue that needs to be addressed differently when debugging

Demonstrate each Error Type

SyntaxError

```
# Missing a colon at the end of the if statement
if True
    print("This will cause a SyntaxError.")
```

```
SyntaxError: expected ':'
```

NameError

```
# Trying to print a variable that hasn't been defined
print(undefined_variable)
```

```
NameError: name 'undefined_variable' is not defined
```

IndexError

```
# Accessing an index that is out of bounds for the list
my_list = [1, 2, 3]
print(my_list[3])
```

```
IndexError: list index out of range
```

TypeError

```
# Adding an integer to a string, which is not allowed
```

```
print("The number is: " + 123)
```

```
TypeError: can only concatenate str (not "int") to str
```

ValueError

```
# Converting a string that does not contain a number to an
int("This is not a number")
```

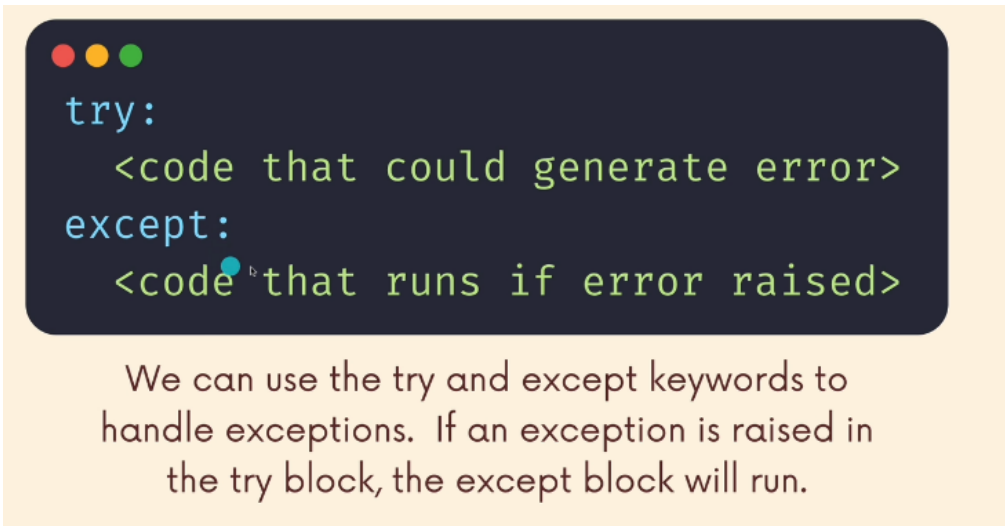
```
ValueError: invalid literal for int() with base 10: 'This is not a number'
```

KeyError

```
# Trying to access a dictionary key that does not exist
my_dict = {'a': 1, 'b': 2}
print(my_dict['c'])
```

```
KeyError: 'c'
```

Try Except: Python's try catch



Generic Except

Except block runs when any type of exception occurs

```
def divide(a, b):
    try:
        return a/b
    except:
        return "You cannot divide by 0"
```

```
5.0
You cannot divide by 0
```

Log the Exception

This works regardless of what type of exception is raised

```
try:
    x = a + b
except Exception as err:
    print(err)
    return {"error": str(err)} # must convert error to stri
```

```
name 'a' is not defined
```

Pass data to Except Block

- When you supply ValueError() with something, it is stored in the exception object's `args` attribute
- The `args` attribute is a tuple that contains all args passed to the exception - one you can take data from using bracket notation + indexes

```
try:
    raise ValueError({'job': "Architect", 'pay': 98000})
```

```
except ValueError as err:
    data = err.args[0]
    print(data) # {'job': 'Architect', 'pay': 98000}
```

Handle Different Kinds of Errors

You can handle different error types through the use of multiple except blocks
Use the generic `Exception` block last or else the others won't get reached

Example 1: Cause NameError block to run

```
def divide(a, b):
    try:
        vari = c + a  # WE CAUSE A NAME ERROR: using c even
        return a / b
    except ValueError as e:
        print("ValueError", e)
    except NameError as e:
        print("NameError", e)
    except Exception as e:  # should go last, or else other
        print("General", e)
divide(9, 0)
```

```
NameError name 'c' is not defined
```

Example 2: Cause ValueError block to run

```
def divide(a, b):
    try:
        int("This is not a number")   # CAUSES VALUE ERROR
        return a / b
    except ValueError as e:
        print("ValueError", e)
    except NameError as e:
        print("NameError", e)
    except Exception as e:  # should go last, or else other
        print("General", e)
divide(9, 0)
```

```
ValueError invalid literal for int() with base 10: 'This is not a number'
```

else block: Executes if no error occurs

If you add an else block after a try/except, the code inside it will run as long as no errors went down

```
def divide(a, b):
    try:
        print("Quotient is", a/b)
    except:  # should go last, or else other except blocks
        print("Generic Error:")
    else:
        print("Function executed with no errors")
divide(9, 1)
```

```
Quotient is 9.0
Function executed with no errors
```

finally block: Executes Regardless of Errors (if any)

If you add a finally block after a try/except, the code inside it will run no matter what
Gets ran if there was an error, or if there wasn't

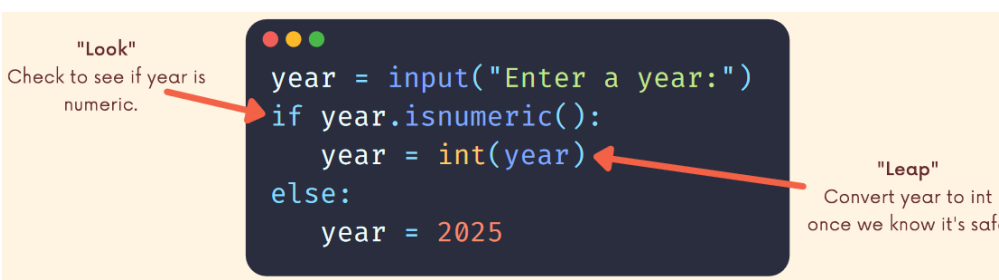
```
def divide(a, b):
    try:
        return a / b
    except Exception as e:  # should go last, or else other
        print("Generic Error:", e)
    finally:
        print("Function completed")
divide(9, 0)
```

```
Generic Error: division by zero
Function completed
```

Error Handling Approaches: LBYL vs EAFP

Look Before you Leap

Test for certain conditions before writing code that could potentially cause errors
Involves lots of conditionals



Easier to ask for Forgiveness than Permission (More popular)

Assume your code will work, then catch exceptions if you're wrong
Involves lots of try except blocks

Assume it'll work
Try converting year to an integer
Catch exception if you're wrong!
This code runs if year can't be cast to an int
try:
year = int(input("Enter a year:"))
except ValueError:
year = 2025

Async Programming w/ Asyncio

Definitions

ASYNC OPERATION
An operation that does not execute immediately
Ex. Fetching data from endpoint or API

Async Await Functions + sleep()

Asyncio is a built-in module used to define async functions in Python

asyncio.sleep(n)

Only use inside of async functions

- This method pauses execution for n seconds and is often used to mock async operations like calling an endpoint
- Use inside of async functions and await the output (shown in previous subsection)

```
await asyncio.sleep(5) # make rest of code wait 5 seconds b
```

Async Function Syntax

- Async functions have the word `async` preceeding them
- Operations that don't execute immediately should have the word `await` in front of them
  This halts code execution until the operation is finished
- To run an async method, use `asyncio.run(asyncFnName())`
  Notice how we call the async function, we do not just namedrop it

Example:

```
import asyncio
async def log_messages():
    print("Start")
    await asyncio.sleep(5) # wait for 5s (halts rest of cod
    print("Finish!")
asyncio.run(log_messages())
```

There are 5 seconds between when "Start" and "Finish" are logged

```
Start
Finish!
```

Tasks Theory

DEFINITION
Running Async Code in a Synchronous Manner
Means running async operations in sequence, 1 after the other as soon as the previous task finishes - which is oftentimes inefficient

Node.js Refresher: Blocking vs Non Blocking Code

REALISTIC EXAMPLE
Imagine you're at a restaurant and there are 2 tables and one waiter (represents single threaded nature of Node.js)

- Blocking behavior: (inefficient)
  Waiter takes order for table 1, waits for meal to be cooked, serves table 1, then moves on to take table 2's order

- Non blocking behavior: (more efficient)
  Waiter takes order for table 1, takes order for table 2, waits for table1's order to be cooked and serves it, then does the same for table 2's order

HOW THIS RELATES TO OUR CODE

- If we simply use `await` in an async function then execute functions in sequence, that is inefficient blocking behavior

We are running async operations in a synchronous manner (1 task after the other)

- We should implement non-blocking behavior and take care of other responsibilities as soon as we have free time
  This is exactly what Asyncio tasks do

Tasks don't Count as Multithreading

Multi threading involves running multiple async operations at the same time, which is not what we accomplish with tasks
Tasks simply hand over control to the event loop, and allow other tasks to run while 1 task is busy waiting for something

TASKS:
Using a single thread to perform async operations in a synchronous manner as optimally as possible

MULTITHREADING:
Performing multiple async operations at the same time

When to use Asyncio Tasks

DO NOT USE WHEN:
The outcome of 1 async operation is needed for the next async operation

```python
import asyncio
# mock API call the gets the name of the user's city
async def fetchCity():
    print("Fetching user's local city...")
    await asyncio.sleep(2)
    return "Toronto"
# mock API call that fetches the population of a city you f
async def fetchCityPopulation(city):
    print(f"Fetching population of {city}...")
    await asyncio.sleep(2)
    return 9000000
# executes async operations in sequence.
# Can't use tasks because we need the city name to fetch th
async def getLocalPopulation():
    # Fetch user's local city
    city = await fetchCity()
    # Use local city to fetch its population, then return
    pop = await fetchCityPopulation(city)
    return pop
result = asyncio.run(getLocalPopulation())
print(f"User has {result} people in their city")
```

```
Fetching user's local city...
Fetching population of Toronto...
User has 9000000 people in their city
```

USE TASKS WHEN:
2 async operations do not rely on each other's results, and they may as well be run in parallel
(tasks don't run things in parallel, but it's a good indicator of when you may as well use it)

```python
import asyncio
# mock API call the gets the name of the user whose id the
async def fetchUserName(id):
    print(f"Fetching name of user #{id}")
    await asyncio.sleep(2)
    return "JasonXQC"
# mock API call the gets the bookmarks of the user whose id
async def fetchUserBookmarks(id):
    print(f"Fetching bookmarks for user #{id}...")
    await asyncio.sleep(2)
    return ["w3schools.com", "chat.openai.com"]
# executes async operations in sequence.
# Should use tasks because fetchUserName and fetchUserBookm
# Both of them only rely on id which we already have (may a
async def getUserData(id):
    # Fetch user's local city
    name = await fetchUserName(id)
    # Use local city to fetch its population, then return
    bookmarks = await fetchUserBookmarks(id)
    return {"username": name, "bookmarks": bookmarks}
result = asyncio.run(getUserData(100))
print(result)
```

```
Fetching name of user #100
Fetching bookmarks for user #100...
{'username': 'JasonXQC', 'bookmarks': ['w3schools.com', 'chat.openai.com']}
```

create_task()

Task Syntax

Inside of an async function...

1. Declare a Task
2. await it
    - Recommend awaiting tasks after other async operations
    - Otherwise there's 0 point to using them - could just await async operations in sequence without tasks

```
task_name = asyncio.create_task(async_operation())
```

When Tasks Run

Asyncio tasks will run while our code waits for other async operations to run their course
We're providing an example that showcases what we mean

- Async operations are mocked using `asyncio.sleep()`
- We run several print statements to showcase where our code execution goes
  It jumps from method to method every time the code gets free time as it needs to wait for `asyncio.sleep()`

DEMONSTRATE

```python
import asyncio
import time
async def printABC():
    print("A")
    await asyncio.sleep(2)
    print("B")
    await asyncio.sleep(2)
    print("C")
async def print123():
    print("1")
    await asyncio.sleep(2)
    print("2")
    await asyncio.sleep(2)
    print("3")
async def printXYZ():
    print("X")
    await asyncio.sleep(2)
    print("Y")
    await asyncio.sleep(2)
    print("Z")
async def main(id):
    start_time = time.time()
    task = asyncio.create_task(printABC())  # Task to run A
    await print123()  # 1 2 3
    await printXYZ()  # X Y Z
    await task # await a task AFTER other async operations
```

```
    print("E")
    print("F")
    print(f"total time: {time.time() - start_time}")
asyncio.run(main(1))
# LOGS IN THE FOLLOWING ORDER: 1 A 2 B 3 X C Y Z E F
```

Order of logs Explained

The order of the logs is very important to understand where the code execution goes
It zips between different methods as soon as it encounters free time waiting for `asyncio.sleep()`

WHAT HAPPENS WHEN TASK IS DEFINED
Task is defined to run `printABC`

- It is awaited after `print123` and `printXYZ` finish running
- This means `printABC` will run when free time is available, but will only finish...
  a) If there's enough free time to do so while `print123` and `printXYZ` run their course
  b) When `print123` and `printXYZ` methods finish running (then `printABC` will be given full attention to run what's left of its course)

LOG ORDER EXPLAINED: `1 A 2 B 3 X C Y Z E F`

1. print123 executes and logs 1, then waits
2. Free time found
   printABC executes and logs A, then waits
3. print123 executes and logs 2, then waits
4. Free time found
   printABC executes the logs B, then waits
5. print123 executes then logs 3, ending the function
6. printXYZ executes then logs X, then waits
7. Free time found
   printABC executes then logs C, ending the function
8. The rest of the logs occur synchronously with no more switching when

free time is found

- print XYZ executes then runs Z then Y, ending the function
- main function execites then runs E, then F

Location Where Task is Awaited Matters

RULE:
Await a task after 1 or more async operations
Otherwise you may as well not use tasks at all - you'll be running async operations 1 after the other in sequence

Example:
Take the previous example and change where task is awaited

```
# rest of code is the exact same (Feel free to copy past ev
async def main(id):
    start_time = time.time()
    task = asyncio.create_task(printABC())  # A B C
    await task # defined BEFORE other async operations (not
    await print123()  # 1 2 3
    await printXYZ()  # X Y Z
    print("E")
    print("F")
    print(f"total time: {time.time() - start_time}") # A B
result = asyncio.run(main(1))
print(result)
# LOG ORDER: # A B C 1 2 3 X Y Z E F
```

gather() : Equivalent to Promise.all

The `asyncio.gather()` method lets you run multiple async operations concurrently, just like how Promise.all does so in JS

- Keep in mind that you aren't running tasks in parallel with `asyncio.gather` or `Promise.all`
- You're allowing Python to switch between tasks when free time is found at await points

Syntax

```
import asyncio
async def asyncOperation1():
    await asyncio.sleep(4)
    return 1
async def asyncOperation2():
    await asyncio.sleep(4)
    return 5
async def asyncOperation3():
    await asyncio.sleep(4)
```

```
    return 10
async def main():
    # Must use await to wait for all async operations to re
    results = await asyncio.gather(asyncOperation1(), async
    print(results)  # asyncio.gather() returns an array of
    # Unpack the array of outputs then return them in an ob
    a, b, c = results
    return {"a": a, "b": b, "c": c}
result = asyncio.run(main())
print(result)
```

```
[1, 5, 10]
{'a': 1, 'b': 5, 'c': 10}
```

Handling Errors

If one of the async operations fed to `asyncio.gather()` raises an exception, it might potentially cancel other tasks
Use error handling on the async_operation methods or the method where you call them

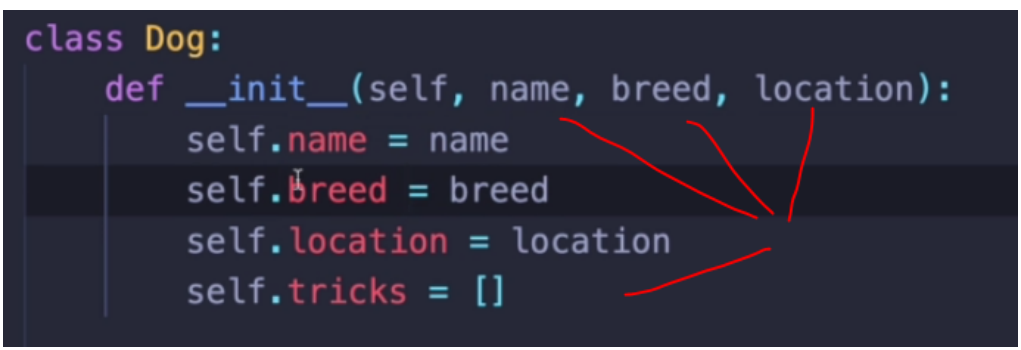
```python
import asyncio
async def asyncOperation1():
    await asyncio.sleep(4)
    return 1
async def asyncOperation2():
    await asyncio.sleep(4)
    return 5
async def asyncOperation3():
    raise ValueError # raise an exception on purpose
    await asyncio.sleep(4)
    return 10
async def main():
    try:
        # Wait for all async operations to resolve
        results = await asyncio.gather(asyncOperation1(), a
        print(results)  # [1, 5, 10]
        # Unpack the array of outputs then return them in a
        a, b, c = results
        return {"a": a, "b": b, "c": c}
    except:
        return None
result = asyncio.run(main())
print(result) # None
```

Classes & OOP

Terminology

INSTANCE:
A Python object that was generated from a class

INSTANCE ATTRIBUTES:
The values you assign to an instance



Intro to OOP

What are Classes

Classes are blueprints that we use to generate Python Objects that we'll refer to as instances
A class describes what properties and functionalities the instances will contain

Practical Use Cases for Classes

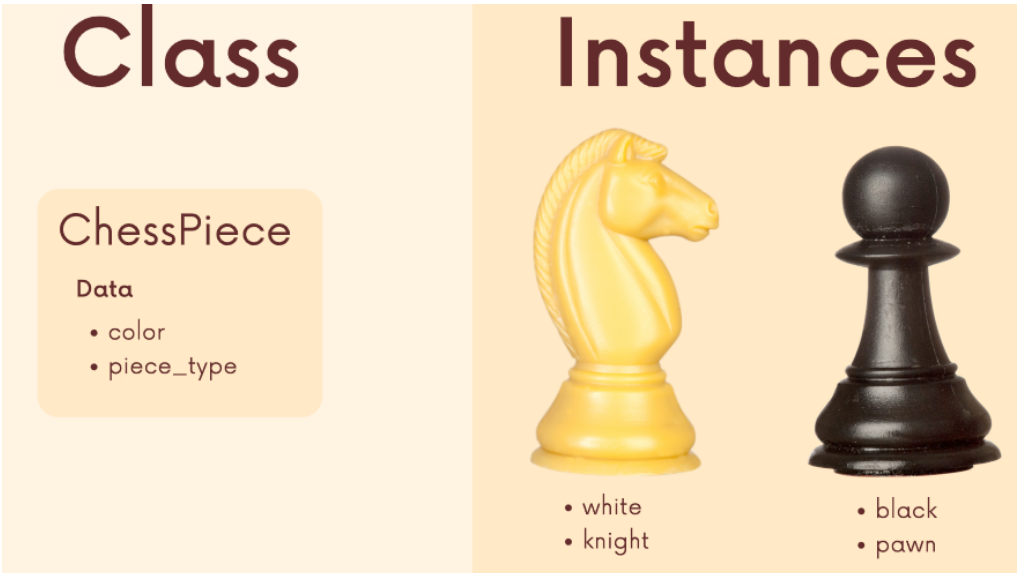
Classes can store both data and functionality, making it useful for modelling things in real life

- Think about how you use functions to automate certain repeated

calculations

- Classes are used to automate the creation of certain archetypes

EXAMPLE: Model chess pieces with classes



Class Syntax

Keep in mind that Python Objects generated from a class are not Dictionaries The most important difference is that you can access and set values on Python objects using dot notation

Class with Hardcoded Instance Attributes

Hardcoded Instance Attributes will be the same for every instance

```
class Game:
    def __init__(self):
```

```
        self.name = "Ruiner"
        self.price = 10.99
# Generate an instance then read its Instance Attributes
g1 = Game()
print(g1.name) # Ruiner
print(g1.price) # 10.99
```

Classes with Dynamic Instance Attributes

Dynamic Instance Attributes need to be set when an instance is initialized

- In practical scenarios, we'll have a mix of hardcoded and dynamic instance attributes
- For example, every game starts off with no review scores upon release That's why we use a hardcoded instance attribute - the empty list

```
class Game:
    def __init__(self, name, price):
        self.name = name
        self.price = price
        self.reviews = [] # all games start off with 0 revi
# Generate an instance then read its Instance Attributes
g1 = Game("Doom", 25.99)
print(g1.name) # Doom
print(g1.price) # 25.99
print(g1.reviews) # []
```

Common Practices

Class Name

- Singular
- Start with capital

self parameter

- Typically named self, to refer to the instance being generated
- Can be named anything, but this is not something you should take advantage of

Instance Attribute Values (IAV)

Instance attribute values are unique to each instance of the class

- They are declared in the `__init__` method of the class declaration
- Often prefaced by `self`

Dot Notation

Use dot notation to edit the Instance Attribute Values

```
class Game:
    category: "entertainment" # class attribute

    def __init__(self):
        self.name = "Ruiner"  # instance attribute value
        self.price = 10.99    # instance attribute value
# Generate an instance then read its Instance Attributes
g1 = Game()
print(g1.name) # Ruiner
print(g1.price) # 10.99
# Change an instance attribute's value
```

```
g1.price = 5.99
print(g1.price) # 5.99
```

Instance Methods

Define a method inside of your class and indent it- on the same level as your `__init__` method

- Each instance method must have the first argument be `self`
  We'll need it to access the instance attribute values
- In professional projects, the logic inside the instance methods will likely be more complex

```
class Game:
    def __init__(self, name, price):
        self.name = name
        self.price = price
        self.reviews = []  # all games start off with 0 rev
    # Define a method that adds to the reviews list
    def add_review(self, score):
        self.reviews.append(score)
    # Define a method that reduces the price of the game
```

```
    def apply_sale(self, percent_discount):
        self.price = self.price * (1 - percent_discount)
# Generate an instance then read its Instance Attributes
g1 = Game("COD Ghosts", 60)
print(g1.price)  # 60
print(g1.reviews)  # []
g1.add_review(4) # add a 4 star review
g1.apply_sale(0.2) # apply a 20% sale
print(g1.price)  # 48
print(g1.reviews)  # [4]
```

When to use each Approach

Class Attributes

Class Attributes: These belong to the class itself.
They are shared by all instances of the class.
If you modify a class attribute, the change is reflected for all existing and future instances.
They are defined *within* the class but *outside* of any method.

Instance Attributes: These belong to a specific instance of the class.
Each instance gets its own copy of these attributes.
Changes to an instance attribute only affect that particular instance. They are typically defined *inside* the `__init__` method (the constructor) using

Access 1 Instance Attribute (Static)

Define a variable inside of a class to define a class attribute
Our reason for using a class attribute for field is because every developer technically works in IT

```
class Developer:
    field = "IT"  # class attribute
    def __init__(self, name, salary):
        self.name = name    # instance attribute value
        self.salary = salary
        self.recommendations = []
# Check the class attribute
print(Developer.field) # IT
# Declare 2 instances, and check their instance attributes
dev1 = Developer("Jason", 71000)
dev2 = Developer("Shadid", 180000)
print(dev1.name) # Jason
print(dev1.name) # Shadid
print(dev2.field) # IT
```

Access 1 Instance Attribute (Dynamic)

You can access an instance attribute using dynamic values using `getattr(instance, field)`

```
class Developer:
    field = "IT"  # class attribute
    def __init__(self, name, salary):
        self.name = name    # instance attribute value
        self.salary = salary
        self.recommendations = []
inst = Developer("Jason", 71000)
```

```
dynamic_field = "salary"
field_value = getattr(inst, dynamic_field) # 71000
```

Access All Instance Attributes

Or you can get a dict containing all instance attributes using `instance.__dict__` or `vars(instance)`

```
class Developer:
    field = "IT"  # class attribute
    def __init__(self, name, salary):
        self.name = name    # instance attribute value
        self.salary = salary
        self.recommendations = []
inst = Developer("Jason", 71000)
all_field_values = inst.__dict__  # {'name': 'Jason', 'sala
all_field_values = vars(inst)    # {'name': 'Jason', 'salary
```

Changing Class Attributes

On the Class (widespread change)

Use dot notation

```
class Developer:
    field = "IT"  # Every developer is in the IT field in s
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        self.recommendations = []
# Check the class attribute using the class itself
print(Developer.field) # IT
# Declare 2 instances
dev1 = Developer("Jason", 71000)
dev2 = Developer("Shadid", 180000)
# Change the class attribute, then check
Developer.field = "Information Technology"
# Check the class attribute on the class and instances
print(Developer.field) # Information Technology
print(dev1.field) # Information Technology
print(dev2.field) # Information Technology
```

```
IT
Information Technology
Information Technology
Information Technology
```

On 1 Instance (isolated change)

You can change a class attribute on an instance, then have it affect only that 1 instance

```python
class Developer:
    field = "IT"  # Every developer is in the IT field in s
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        self.recommendations = []
# Check the class attribute using the class itself
print(Developer.field) # IT
# Declare 2 instances
dev1 = Developer("Jason", 75000)
dev2 = Developer("Shadid", 180000)
# Change the class attribute on the dev1 instance only
dev1.field = "Information Technology" # HERE
# Check the class attribute on the class and instances
print(Developer.field) # IT
print(dev1.field) # Information Technology
print(dev2.field) # IT
```

```
IT
IT
Information Technology
IT
```

Bad Practice: Change Class Attributes with Instance Methods

This method is considered a bad practice since any instance will be able to

change a property that's universally shared
This is not great since class attributes are meant to hold values that are universal standards across all instances

```
class MyClass:
    class_attribute = "I am a class attribute"
    def __init__(self, instance_attribute):
        self.instance_attribute = instance_attribute
    # Define an instance method that edits a class attribut
    def change_class_attribute(self, new_value):
        MyClass.class_attribute = new_value
```

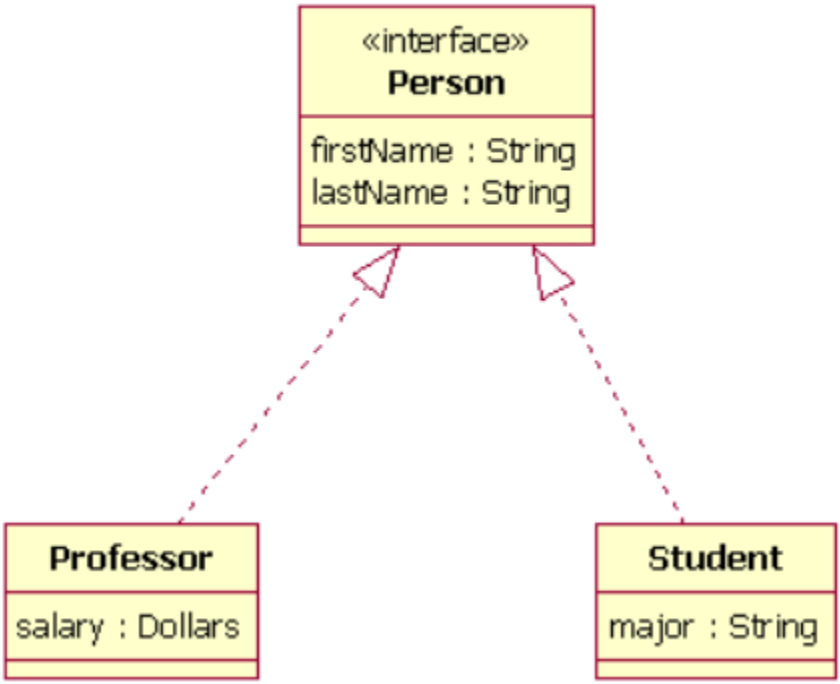
Overview: Instance Attribute Values VS Class Attributes

Class Inheritance + super()

Concept behind Class Inheritance

A class can inherit instance attributes and methods from another class
In real life, there are many models that could possibly have submodels

- Professors and Students are both a type of person
- They both have a `firstName` and `lastName`, plus unique attributes of their own
  This is why the `Professor` and `Student` classes inherit from the Person class



Subclass without its own Dynamic IAVs

SHOWCASE CLASS INHERITANCE

- Notice how the WebDeveloper class generates instances that share the greeting method and name IAV
- That's because the WebDeveloper class inherits from the Programmer class

NEXT LESSON:
If you want to create a class that has dynamic instance attribute values, you must use the super() method

```
class Programmer:
    def __init__(self, name):
        self.name = name
    # Define an instance method that edits a class attribut
    def greeting(self):
        print("Hello world!")
class WebDeveloper(Programmer):
    def introduce_self(self):
        print(f"I'm {self.name}, and I'm a web dev!")
# Generate instance
myself = WebDeveloper("Jason")
print(myself.name) # Jason
myself.greeting() # Hello world!
myself.introduce_self() # I'm Jason, and I'm a web dev!
```

Subclass with its own Dynamic IAV's

If you want to create a class that has dynamic instance attribute values, you must use the `super()` method
The `super()` method is used to refer to the base/parent class

- Without super(), we could not have passed "MERN stack" to the WebDeveloper class
- We could have only passed "Jason" for the name

```
class Programmer:
    def __init__(self, name):
        self.name = name
    def greeting(self):
        print("Hello world!")
class WebDeveloper(Programmer):
    def __init__(self, name, stack):
        # Use super() to pass values to the __init__ functi
        super().__init__(name)
        self.stack = stack
    def shout_stack(self):
        print(f"I use {self.stack}")
# Generate instance
myself = WebDeveloper("Jason", "MERN stack")
print(myself.name) # Jason
myself.greeting() # Hello world!
myself.shout_stack() # I use MERN stack
```

Inheriting fr/ Multiple Parents

A class can inherit IAV's and methods from more than 1 parent class

Example 1: no super()

```
class Parent1:
    def method1(self):
        print("Method from Parent1")
class Parent2:
    def method2(self):
        print("Method from Parent2")
class Child(Parent1, Parent2):
    def child_method(self):
        print("Method from Child")
# Create an instance of the Child class
child_instance = Child()
# Access methods from both parent classes and the child cla
child_instance.method1()  # Method from Parent1
child_instance.method2()  # Method from Parent2
child_instance.child_method()  # Method from Child
```

Example 2: With super()

```
class Parent1:
    def __init__(self, input_value):
        self.value1 = input_value
        print(f"Parent1 initialized with value: {self.value
    def method1(self):
        print("Method from Parent1")
class Parent2:
    def __init__(self, input_value):
        self.value2 = input_value
        print(f"Parent2 initialized with value: {self.value
    def method2(self):
        print("Method from Parent2")
class Child(Parent1, Parent2):
    def __init__(self, input_value_parent1, input_value_par
        # Call the constructors of the parent classes in th
        super().__init__(input_value_parent1)     # pass val
        Parent2.__init__(self, input_value_parent2)  # Pass
        self.value_child = input_value_child
        print(f"Child initialized with value: {self.value_c
    def child_method(self):
        print("Method from Child")
# Create an instance of the Child class
child_instance = Child("Value1", "Value2", "ChildValue")
# Access methods and attributes from all classes
child_instance.method1()  # Method from Parent1
child_instance.method2()  # Method from Parent2
child_instance.child_method()  # Method from Child
```

```
Parent1 initialized with value: Value1
Parent2 initialized with value: Value2
Child initialized with value: ChildValue
Method from Parent1
Method from Parent2
Method from Child
```

Why super() refers to Parent1 in Prev. Example

The `super()` function is used to refer to the next class in the method resolution order (MRO). In multiple inheritance scenarios, when a class inherits from more than one parent class, Python needs a mechanism to determine the order in which the parent classes are called. This order is defined by the method resolution order.

In the `Child` class, `super().__init__(input_value_parent1)` is used to call the constructor of the next class in the MRO after `Child`, which is `Parent1`. It's a way to explicitly invoke the constructor of the immediate parent class.

However, in the case of `Parent2`, the `super()` function is not used because `Parent2.__init__(self, input_value_parent2)` is used instead. This approach directly calls the constructor of `Parent2` without relying on `super()`. In this specific example, it would work the same way if you used `super()` for `Parent2` as well, but it's worth noting that in a more complex inheritance hierarchy, using `super()` consistently is a good practice to ensure that the MRO is followed correctly.

In summary, `super()` is used to call the constructor of the immediate parent class, and the specific parent class is mentioned explicitly when using direct calls, such as `Parent2.__init__(self, input_value_parent2)`.

Read Only Attributes

In Python, you can create read-only attributes in classes by using property decorators.
The `property` decorator allows you to define a method as if it were an attribute, and it can be used to control access to the attribute.

```
class ReadOnlyClass:
    def __init__(self, value):
        self._read_only_attribute = value
    @property
    def read_only_attribute(self):
        return self._read_only_attribute
# Creating an instance of the class
obj = ReadOnlyClass("Hello, this is read-only!")
# Accessing the read-only attribute
print(obj.read_only_attribute)  # Output: Hello, this is re
# Attempting to modify the read-only attribute will raise a
try:
    obj.read_only_attribute = "Trying to modify!"
except AttributeError as e:
    print(f"AttributeError: {e}")
```

Turn Class Instances into Dicts

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
```

```
obj = ListNode(6)
obj.claude = 8
print(obj.__dict__) # {'val': 6, 'next': None, 'claude': 8}
```

< Class Methods >

https://www.udemy.com/course/one-week-python/learn/lecture/31682872#overview

= = = = = = = **REMAINING**

= = = = = = =

< Random Methods >

type(): Identifies Type of value

How it Works

Identifies the type of a value

```
y = type("65")
print(y)    # <class 'str'>
print(type(5))   # <class 'int'>
```

isInstance is Better

BETTER TYPE CHECKING METHODS EXIST
isInstance is typically used over type() when it comes to writing conditional logic that deals with value types

< Bit Manipulation >

< Bitwise OR Operator: != >

FROM CHAT GPT

The `|=` operator in Python is an in-place bitwise OR assignment operator. It performs a bitwise OR operation between the current value of the variable on the left-hand side and the value on the right-hand side, and then assigns the result back to the variable on the left-hand side.

Here is an example to illustrate its usage:

```
a = 5  # binary: 0101
b = 3  # binary: 0011
a |= b  # equivalent to a = a | b
```

After the `a |= b` operation, the binary representation of `a` and `b` will be ORed together:

```
0101
| 0011
-------
  0111
```

The result, `0111`, is `7` in decimal, so `a` will be updated to have the value `7`.

It's important to note that the `|=` operator can also be used with sets in Python, where it performs an in-place union operation:

```
a = {1, 2, 3}
b = {3, 4, 5}
a |= b  # equivalent to a.update(b)
```

After this operation, `a` will be `{1, 2, 3, 4, 5}`, as it's updated to be the union of `a` and `b`.

Bitwise Operators

https://www.w3schools.com/python/python_operators.asp#:~:text=Python%20Bitwise%20Operators

< Multi-threading >

< Multi-processing >

Return2

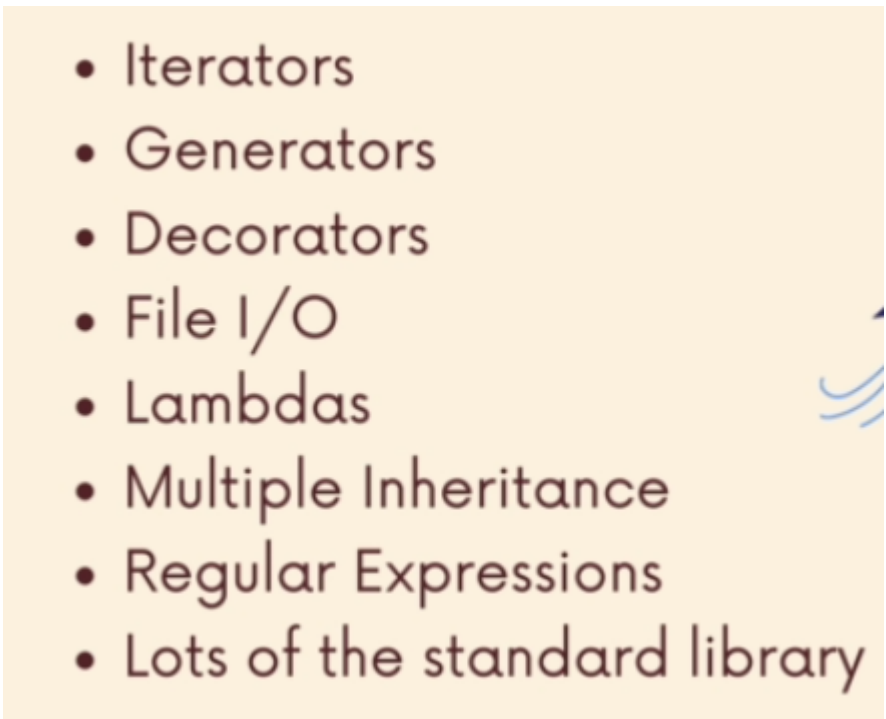
Understand all Python Data Types

isNumeric

THREADING
https://www.youtube.com/watch?v=A_Z1lgZLSNc

Next Steps

Remaining Topics



Dates
https://www.w3schools.com/python/python_datetime.asp

< Identity Operators >

Repeated at JUMP440 (provide summarized notes at other side of warp)
Can be used with both lists and dictionaries I think

is	Evaluates to True if a and b both refer to the same object in memory
is not	Evaluates to True if a and b do NOT refer to the same object in memory